z/OS

# DCE Application Development Guide: Core Components

SC24-5905-00

z/OS

# DCE Application Development Guide: Core Components

SC24-5905-00

**Note**

Before using this information and the product it supports, be sure to read the general information under Appendix C, “Notices” on page 553.

## First Edition (March 2001)

This edition, SC24-5905-00, applies to Version 1 Release 1 of z/OS DCE Base Services, z/OS DCE user Data Privacy (DES and CDMF), z/OS DCE User Data Privacy (CDMF) (program number 5694-A01), and to all subsequent releases and modifications until otherwise indicated in new editions.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

IBM welcomes your comments. A form for reader's comments may be provided at the back of this publication, or you may address your comments to the following address:

International Business Machines Corporation
Information Development, Dept. G60
1701 North Street
Endicott, NY 13760-5553
United States of America

FAX (United States & Canada): 1+607+752-2327
FAX (Other Countries):
Your International Access Code +1+607+752-2327

IBMLink™ (United States customers only): GDLVME(PUBRCF)
Internet e-mail: pubrcf@vnet.ibm.com
World Wide Web: http://www.ibm.com/servers/eserver/zseries/zos/

If you would like a reply, be sure to include your name, address, telephone number, or FAX number.

Make sure to include the following in your comment or note:

- Title and order number of this book
- Page number or topic related to your comment

When you send information to IBM, you grant IBM a nonexclusive right to use the information in any way it believes appropriate without incurring any obligation to you.

The following statements are provided by the Open Software Foundation.

# Contents

# Figures

# Tables

# About This Book

The objective of this book is to assist you in designing, writing, compiling, linking, and running distributed applications on the IBM z/OS operating system using z/OS DCE. The steps to develop a distributed application using DCE services and application programming interfaces (API) are described in progressive detail. Also discussed are the development decisions and tools that you need to consider when developing your distributed application using z/OS DCE.

To create DCE applications that access IMS™ or CICS® transactions, refer to *z/OS DCE Application Support Programming Guide*.

## Who Should Use This Book

This book assumes you are an experienced application developer or programmer with a working knowledge of the C programming language and the z/OS operating system. You do not have to possess prior knowledge of, or experience with, designing and writing distributed applications using the Open Software Foundation (OSF) Distributed Computing Environment (DCE) services and APIs.

Ideally, you should be able to:

- Allocate z/OS data sets
- Edit, browse, and copy z/OS data sets and associated members
- Print data sets
- Write and submit batch jobs on z/OS
- Write, compile, link, and run C/C++ programs on z/OS
- Write and understand JCL to run on z/OS
- Understand Shell and TSO/E commands.

A good working knowledge and understanding of the following would be helpful:

- Interactive System Productivity Facility/Program Development Facility (ISPF/PDF)
- Concepts behind a distributed application
- Using the Spool Display and Search Facility (SDSF) to check on the status of your application.

Some exposure to the UNIX or AIX® operating system is helpful but not essential to use this book.

You should be familiar with the concepts of the Distributed Computing Environment. If you are not, read *z/OS DCE Introduction*.

## DCE Application Development Environment

It is conceivable that you may develop your DCE applications on a platform other than the z/OS operating system. Perhaps you may prefer to work on a UNIX-based workstation or a proprietary operating system. If your goal is to ultimately run either the client or server portion of your DCE application on z/OS, ensure that portion of your DCE application conforms to all recommendations contained in this book.

This book describes the development steps assuming you are developing your DCE applications on the z/OS operating system. If you are developing DCE applications on the z/OS platform that are targeted to run on another platform, consult the DCE application development documentation associated with that platform.

## Unsupported OSF DCE Functions

The following DCE technology functions, which may be available in the Distributed Computing Environment product from OSF or on DCE offerings from other vendors, are not supported in z/OS DCE:

- DCE Directory Services
  - X/Open Data Services (XDS) function (Global Directory Service (GDS) portion)
  - X/Open OSI-Abstract-Data Manipulation (XOM) function (GDS portion)
  - Global Directory

On z/OS, only CDS, XDS, and XOM access to CDS are supported. GDS, XDS, and XOM access to GDS are not supported.

The following DCE daemon is not supported on z/OS DCE:

- DCE Security daemon

  **Note:** Although the DCE Security daemon is not included in z/OS DCE, a security server is available from IBM as a separately licensed program.

## OSF DCE Programming Interfaces

- pthread interfaces
  - The following interfaces are not supported by z/OS DCE and return -1, errno ENOSYS:
    - pthread_attr_getinheritsched()
    - pthread_attr_getprio()
    - pthread_attr_getsched()
    - pthread_attr_setinheritsched()
    - pthread_attr_setprio()
    - pthread_attr_setsched()
    - pthread_getprio()
    - pthread_getscheduler()
    - pthread_setprio()
    - pthread_setscheduler()
  - For all pthread interfaces (including mutexes, threads, condition variables and so on), the interfaces do not accept copies of the objects as a parameter. The object returned from the pthread interface to create the object must be used at all times.
  - Unlike the OSF DCE implementation, the z/OS DCE implementation of the following functions can raise an exception (**exc_e_cpa_error**) in error situations:
    - pthread_lock_global_np()
    - pthread_unlock_global_np()
  - pthread_cond_timedwait() expects an absolute hardware time (that is, time-of-day clock value) for the wait time instead of the DCE software clock time, which is what OSF/DCE expects. pthread_get_expiration_np() returns a software adjusted time as in the OSF/DCE model, and is used as input to pthread_cond_timedwait().
  - exc_report() does not print out a message to stderr as expected. z/OS DCE uses Reliability, Availability and Serviceability (RAS) services to log messages instead of this function.
  - **pthread_cond_init** cannot initialize a condition variable more than once.
  - **pthread_mutex_init** cannot initialize a mutex more than once.
- Exceptions

  - z/OS DCE catches z/OS ABENDs in addition to the set of predefined exceptions and user defined exceptions.
  - TRY/CATCH/ENDTRY macros can raise an exc_e_insfmem exception if they cannot get enough heap storage.
  - TRY/CATCH/ENDTRY macros can raise an exc_e_uninitexc exception if they detect that the CATCH does not specify a valid exception.
- Remote Procedure Call

  The following interfaces are not supported:

  - rpc_mgmt_set_server_stack_size()
- Security Services

  The following interfaces are not supported:

  - sec_login_get_pwent()
  - sec_login_init_first()

## How This Book Is Organized

This guide is divided into five major parts. The first part introduces some DCE facilities and the following parts contain detailed information on using the various DCE components and their respective APIs that are supported by z/OS DCE.

- Part 1, "DCE Facilities" on page 1 introduces two DCE facilities, DCE Host Services and the DCE Backing Store. Descriptions of these facilities and how you might put these useful facilities to work in your applications is detailed in this part.
- Part 2, "Using the DCE Remote Procedure Call APIs" on page 35 shows you how to use the Remote Procedure Call (RPC) APIs to create DCE RPC applications. You are introduced to the RPC model and RPC components, and shown the steps in developing an RPC application. Advanced RPC issues, such as using the Name Service Interface and handling remote errors, are discussed. In addition, the syntax notation conventions and language elements of the Interface Definition Language (IDL) and Attribute Configuration Language (ACF) are described in detail.
- Part 3, "Using the DCE Threads APIs" on page 313 shows you how to increase the performance of your distributed applications using the DCE Threads APIs. You are introduced to Threads concepts and shown several models for multithreaded programming. In addition, you are shown how to use the DCE Threads exception handling interface to handle abnormal conditions in your applications. A comparison between DCE Threads concepts and z/OS multitasking is provided to help you avoid pitfalls caused by semantic differences.
- Part 4, "Using the DCE Distributed Time Service APIs" on page 367 shows you how to use the DTS APIs in DCE applications to convert between different time formats and representations to determine event sequencing, duration, and scheduling. In addition, you are shown how to use external time-provider services with the DCE Time-Provider Interface. A programming example shows you how to use the DTS APIs.
- Part 5, "Using the DCE Security APIs" on page 395 provides details on the DCE Security services and facilities, and describes the main Security interfaces. Walkthroughs of Authentication and Authorization are presented to enhance your understanding of the DCE security concepts. You are shown how to use the DCE Security APIs to enable your client applications to communicate with the Registry server, establish a login context, manage secret keys, and communicate with the Access Control List facility.

To find more information on topics related to application development not addressed in this book, consult the following:

- *z/OS DCE Application Development Reference*, SC24-5908
- *z/OS DCE Administration Guide*, SC24-5904
- *z/OS DCE Application Support Programming Guide*, SC24-5902 (CICS and IMS)
- *z/OS DCE Messages and Codes*, SC24-5912

For example, the DCE CDS is discussed in detail as a separate component in the administration documentation. Similarly, certain aspects of the DCE Security Service important to application developers (such as adding new principals to the registry database) are found only in the administration books.

## Terminology Used in This Book

Because DCE technology has been developed from the UNIX environment, many DCE concepts and terms contained herein relate to that environment. z/OS terms and concepts are used throughout this book wherever possible.

The following table explains how certain terms are used in this book and how they are related.

| Related Terms | Relationship |
|---|---|
| **file**<br>**data set**<br>**sequential data set**<br>**partitioned data set member**<br>**hierarchical file system (HFS) file** | Throughout this book, the term *file* can refer to a sequential data set, a member of a partitioned data set, or a hierarchical file system (HFS) file (UNIX System Services file system). For more information on hierarchical file systems in z/OS, see *z/OS UNIX System Services User's Guide*, SA22-7801. |
| **user prefix**<br>**data set names** | The term *user prefix* is used throughout this book when referring to the names of data sets in a TSO/E environment. In that environment, the user prefix is usually a user's logon identification. If desired, you can set the user prefix to a value other than the your logon identification by using the TSO/E PROFILE command. In z/OS batch mode, your user prefix depends on whether Resource Access Control Facility (RACF®), a component of the SecureWay® Security Server for z/OS, or another security product is installed on your system. If RACF is installed, and you are processing in batch mode, your user prefix can be the same as your logon user identification. If RACF is not installed and you are processing in batch mode under z/OS, you may not have to use a prefix. See your systems programmer to determine the RACF settings for your site.<br><br>Unless otherwise specified, when the full name of a data set is referred to, the high-level qualifier for that data set will be represented by *USERPRFX*. The *USERPRFX* is determined by the application developer, and depends on the library where the application is installed. For example, USERPRFX.EXAMPLE.C(MEMBER) represents a partitioned data set whose first-level qualifier is represented by USERPRFX, whose second-level qualifier is EXAMPLE, and whose third-level qualifier is C. Its member is MEMBER. |
| **application programming interface** (API)<br>**call**<br>**function**<br>**routine** | Throughout this book, the terms *API*, *call*, *function*, and *routine* all refer to the same z/OS DCE application programming interface. For example, **rpc_binding_free()** API, **rpc_binding_free()** call, and **rpc_binding_free()** routine, all refer to the same **rpc_binding_free()** function. |

| Related Terms | Relationship |
|---|---|
| DCE components | Throughout this book, all references to individual DCE components (such as RPC) refer to that component within z/OS DCE. For example, references to RPC, DCE RPC, and z/OS DCE RPC all refer to the same z/OS DCE component. |
| z/OS SecureWay Security Server DCE | In this book the term "DCE Security Server" (or simply "Security Server") refers to the z/OS SecureWay Security Server DCE or to a DCE Security Server provided on another host in the DCE cell. The z/OS SecureWay Security Server DCE is a component of the SecureWay Security Server for z/OS. |
| **daemon**<br>**process**<br>**started task**<br>**address space** | The term *daemon* (originating from the UNIX operating system) is used throughout this book. It is synonymous with a process. Usually there is one process per address space; however, the DCEKERN started task is an exception in that its address space contains several processes (or daemons). |

## Conventions Used in This Book

This book uses the following typographic conventions:

| | |
|---|---|
| **Bold** | **Bold** words or characters represent system elements that you must enter into the system literally, such as commands, options, or path names. |
| *Italic* | *Italic* words or characters represent values for variables. |
| `Example font` | Examples and information displayed by the system appear in `constant width type style`. |
| [ ] | Brackets enclose optional items in format and syntax descriptions. |
| { } | Braces enclose a list from which you must choose an item in format and syntax descriptions. |
| \| | A vertical bar separates items in a list of choices. |
| < > | Angle brackets enclose the name of a key on the keyboard. |
| ... | Horizontal ellipsis points indicate that you can repeat the preceding item one or more times. |
| \ | A backslash is used as a continuation character when entering commands from the shell that exceed one line (255 characters). If the command exceeds one line, use the backslash character **\** as the last non-blank character on the line to be continued, and continue the command on the next line. |

This book uses the following keying conventions:

| | |
|---|---|
| <**Alt-***c*> | The notation <**Alt-***c*> followed by the name of a key indicates a control character sequence. |
| <**Return**> | The notation <**Return**> refers to the key on your keyboard that is labeled with the word Return or Enter, or with a left arrow. |
| **Entering commands** | When instructed to enter a command, type the command name and then press <**Return**>. |

## Where to Find More Information

Where necessary, this book references information in other books using shortened versions of the book title. For complete titles and order numbers of the books for all products that are part of z/OS, see the *z/OS Information Roadmap*, SA22-7500. For complete titles and order numbers of the books for z/OS DCE, refer to the publications listed in the "Bibliography" on page 577.

For information about installing z/OS DCE components, see the *z/OS Program Directory*.

## Softcopy Publications

The z/OS DCE library is available on a CD-ROM, *z/OS Collection*, SK3T-4269. The CD-ROM online library collection is a set of unlicensed books for z/OS and related products that includes the IBM Library Reader.™ This is a program that enables you to view the BookManager® files. This CD-ROM also contains the Portable Document Format (PDF) files. You can view or print these files with the Adobe Acrobat reader.

## Internet Sources

The Softcopy z/OS publications are also available for web-browsing and for viewing or printing PDFs using the following URL:

```
http://www.ibm.com/servers/eserver/zseries/zos/bkserv/
```

You can also provide comments about this book and any other z/OS documentation by visiting that URL. Your feedback is important in helping to provide the most accurate and high-quality information.

## Using LookAt to Look up Message Explanations

LookAt is an online facility that allows you to look up explanations for z/OS messages. You can also use LookAt to look up explanations of system abends.

Using LookAt to find information is faster than a conventional search because LookAt goes directly to the explanation.

LookAt can be accessed from the Internet or from a TSO command line.

You can use LookAt on the Internet at:

```
http://www.ibm.com/servers/eserver/zseries/zos/bkserv/lookat/lookat.html
```

To use LookAt as a TSO command, LookAt must be installed on your host system. You can obtain the LookAt code for TSO from the LookAt Web site by clicking on the **News and Help** link or from the *z/OS Collection*, SK3T-4269.

To find a message explanation from a TSO command line, simply enter: **lookat** *message-id* as in the following:

```
lookat iec192i
```

This results in direct access to the message explanation for message IEC192I.

To find a message explanation from the LookAt Web site, simply enter the message ID and select the release with which you are working.

**Note:** Some messages have information in more than one book. For example, IEC192I has routing and descriptor codes listed in *z/OS MVS Routing and Descriptor Codes*, SA22-7624. For such messages, LookAt prompts you to choose which book to open.

## Accessing Licensed Books on the Web

z/OS licensed documentation in PDF format is available on the Internet at the IBM Resource Link site:

`http://www.ibm.com/servers/resourcelink`

Licensed books are available only to customers with a z/OS license. Access to these books requires an IBM Resource Link user ID, password, and z/OS licensed book key code. The z/OS order that you received provides a memo that includes your key code.

To obtain your IBM Resource Link user ID and password, logon to:

`http://www.ibm.com/servers/resourcelink`

To register for access to the z/OS licensed books:

1. Logon to Resource Link using your Resource Link user ID and password.
2. Select **User Profiles** located on the left-hand navigation bar.
3. Select **Access Profile**.
4. Select **Request Access to Licensed books**.
5. Supply your key code where requested and select the **Submit** button.

If you supplied the correct key code you will receive confirmation that your request is being processed.

After your request is processed you will receive an e-mail confirmation.

**Note:** You cannot access the z/OS licensed books unless you have registered for access to them and received an e-mail confirmation informing you that your request has been processed.

To access the licensed books:

1. Logon to Resource Link using your Resource Link user ID and password.
2. Select **Library**.
3. Select **zSeries**.
4. Select **Software**.
5. Select **z/OS**.
6. Access the licensed book by selecting the appropriate element.

# Part 1. DCE Facilities

This part describes two DCE facilities:

- DCE Host Services
- DCE Backing Store

# Chapter 1. Introduction to DCE Facilities

By now you are aware that DCE consists of a number of major components, each of which addresses some necessary aspect of distributed computing:

- Threads make programs more efficient by allowing parallel execution of portions of code
- Remote Procedure Calls (RPCs) hide network details from applications
- Time Service gives consistent time to widely scattered cells and hosts
- Security gives programs assurances that users and other programs are who they say they are and that they are authorized to do what they are supposed to do
- Directory Service helps clients find servers and other resources

For most applications, a DCE component is not used by itself but the components all work together to create a very useful and powerful environment.

The more you understand DCE and its components, the more you will realize that a strict division by component is not always clear. The document set for DCE is organized by component mostly for the convenience of people trying to explain and understand DCE, but applications often contain a blend of aspects of all the components. This is why it often seems like the information you need to do your work is scattered across many chapters or volumes, and why advanced or unusual features seem to be described along side basic or typical tasks. DCE also has some special facilities that just do not fit neatly into any one discussion of a DCE component, and these are the facilities we describe in this first part of this book.

You should read the *z/OS DCE Application Development Guide: Introduction and Style* prior to using the DCE facilities described here, and you may want to skip to other parts of this book before learning details about the DCE facilities.

For the most part, the DCE facilities are already used by one or more major components of DCE to accomplish some feature they require, while others are stand-alone facilities intended to make developing distributed applications easier. These facilities are described separately here so that you can use them for your own applications too. The DCE facilities include the following:

| | |
|---|---|
| Host Services | Host Services give remote access to several kinds of data and functionality with respect to each DCE host and its servers. Each host runs a DCE Host daemon (**dced**) as the interface to the Host Services. In many cases **dced** automatically maintains the data and performs the functions. Some of the data that you can access (and maintain) remotely includes the host name, the host's cell name, configuration and execution data for all servers on the host, and a database of endpoints (server addresses) through which running servers can be contacted. Some of the functions that you can perform remotely include starting and stopping servers. |
| Backing Store Database Service | You use a backing store to maintain typed data between invocations of applications. For example, you could store application-specific configuration data in a backing store, and then when the application restarts, it could read the previous configuration from the backing store. Data is stored and retrieved by a UUID or character string key, and each record (or data item) may have a standard header if you wish. |

As DCE has developed and improved, useful facilities have been added to make DCE easier and more useful. Some solutions developed to implement a major component's feature can also prove useful to your applications. For example, the Security component must have a way to maintain Access Control Lists (ACLs). While the backing store was developed to handle this kind of task, you can use this facility to store your own application-specific data across invocations.

This first part of this book describes how you might put these useful facilities to work in your applications.

# Chapter 2. DCE Host Services

Every DCE host must maintain certain kinds of data about itself and the servers it provides. For example, each host stores configuration data about its DCE environment, and it also stores data about servers registered and running on the host. In addition, each host needs some services to not only manage this data, but also to administer the host and DCE servers. For example, a service that can start and stop specific servers has obvious value. The DCE host services consist of the following:

| | |
|---|---|
| Endpoint Mapper | The endpoint mapper service enables a client to find servers on a particular host and the services and objects provided by those services. This service maintains on each host an endpoint map that contains a mapping of port addresses (endpoints) to servers, the services servers provide, and the objects servers manage. |
| Host Data Management | The host data management service stores and controls access to such data as the host's cell name, the host name, and the cell alias names, among other things. |
| Server Management | The server management service can start and stop specified servers on a host, enable or disable specific services provided by a server, and manage configuration and execution data about these servers. |
| Security Validation | The security validation service maintains a login context for the host's identity of itself, maintains the host principal's keys, and assures applications (especially login programs) that the DCE Security Daemon (**secd**) is genuine. |
| Key Table Management | A server uses private keys for its security instead of human-entered passwords. The key table management service can be used to manage the keys stored in key tables on a server's host. |

Of course in a distributed environment, these data and services must be easily yet securely accessible from other hosts. The DCE Host Daemon (**dced**) is a continuously running program on each host that provides access to the host services either locally on that host, or remotely from another host.

## Types of Applications

Although your applications may need some aspect of these host services (control over which services are enabled for a particular server, for example), typical servers do not have to do any special coding for them. This reduces the size and complexity of server code, and it keeps the details of administration out of applications. This also takes the burden of server administration off of you so you can concentrate on the application's business functionality.

System administrators will appreciate this development model too because it is unlikely that many servers implementing their own administrative mechanisms will all behave in the same manner. Administrators commonly use the DCE control program, **dcecp**, to access the host services (via **dced**) of any host in their distributed environment (provided the user has the appropriate permissions). The **dcecp** also uses a script language for more sophisticated administration. See the *z/OS DCE Administration Guide* for more on using **dcecp** to access the host services.

Although the **dcecp** commands offer an administrator a great deal of control over DCE hosts and servers, a set of APIs are also supplied for application developers who need to access the DCE host services from an application rather than from scripts or the operating system's command line.

Typical business applications do not use the APIs of these services, but a *management application* might. A management application is a client or server that manages other servers or some aspect of the distributed environment. (The **dced** is itself a management application that is built into DCE.) Some other types of applications that might use these APIs include:

- Applications that control other servers for load balancing or server redundancy.
- Applications that use a graphical user interface (GUI) instead of the command line interface provided by the **dcecp** program.
- Applications that need to monitor a server's current state. For example, an application may need to make sure a particular server or one of its services is available.

## Issues of Distributed Applications

The most important aspect of **dced** is that it gives system administrators the ability to remotely manage services, servers, endpoints, and even objects on any host in DCE. This eliminates the frustrating and tedious task of logging in to many different hosts to manage them. This also allows for scalability because it is impractical to manage a large system by logging in to all its hosts.

The features of **dced** are greatly enhanced when used remotely. Of course an administrator can use **dced** to locally manage a host's services, but **dced**'s real power is in remotely managing system and application server configurations, key tables, server startup, login configurations, and cell information.

Security becomes a major issue when it comes to remote services. With the power of **dced**'s services and the **dcecp** tool, it is important that only authorized principals can use them. The **dced** controls access to its various objects using ACLs. Server keys are security-sensitive data that must be seldom transmitted over the network. All key table data are encrypted when they are transmitted for secure remote key table management.

Finally, the remote capabilities of the **dced** give you real-time status of processes and services in DCE.

## Managing a Host's Endpoint Map

Each DCE host has an endpoint map that contains a mapping of servers to endpoints. Each endpoint map server entry is associated with an array of services (interfaces) provided by the server, and each service is associated with an array of objects supported by the service.

When a typical server calls the **dce_server_register()** routine, the RPC runtime generates the endpoints on which the server will listen for calls and then uses **dced**'s endpoint mapper service of the local host to register the endpoints. Later, when a typical client makes a remote procedure call, its RPC runtime uses the server host's endpoint mapper service to find the server. When the typical server shuts down, it calls the **dce_server_unregister()** routine to remove its endpoints from the endpoint map so that clients do not later try to bind to it.

Applications can also use the lower level **rpc_ep_register()** and associated RPC routines. Because the endpoint map is essential for RPCs to work, endpoints are fully described in Chapter 5, “RPC Fundamentals” on page 59, and the endpoint map structure is described with respect to routing of RPCs in Chapter 10, “Topics in RPC Application Development” on page 173.

The endpoint map is for the most part maintained automatically by **dced**. For example, it periodically removes stale endpoints so that the RPC runtime will not try to complete a binding for a client to a server that is no longer running. However, administrative applications may find it necessary to peruse a remote endpoint map and even remove specific endpoints from a local host's endpoint map.

To read the elements of a remote endpoint map, applications use a loop with the set of routines **rpc_mgmt_ep_elt_inq_begin()**, **rpc_mgmt_ep_elt_inq_next()**, and **rpc_mgmt_ep_elt_inq_done()**. The inquiry can return all elements until the list is exhausted, or the inquiry can be restricted to return elements for the following:

- Elements matching an interface identifier (UUID and version number)
- Elements matching an object UUID
- Elements matching both an interface identifier and object UUID

Administrators can manage the endpoint map by using **dcecp** with the **endpoint** object. In an extreme situation, you could permanently remove endpoints directly from the local endpoint map by calling the **rpc_mgmt_ep_unregister()** routine. This function cannot be done remotely for security reasons.

## Binding to the dced Services

When you write a program that uses a host service, you begin by creating a dced binding to the service on a particular host. Bindings are relationships between clients and servers that allow them to communicate. A dced binding is a specific kind of binding that not only gives your application a binding to the **dced** server, but it also associates the binding with a specific host service on that server.[1]

In general, an application follows these basic steps to use a host service:

1. Establish a binding to the service on the desired host. For example, your application can establish a binding to the host data management service on another host.
2. Obtain one or more **dced** entries for that service. For example, your application can obtain the host data entry that identifies the host's cell name, among other things. This step is valid for the following services:
   - Host data management
   - Server management
   - Key table management

   Depending on the service and function desired, this step may or may not be necessary. For example, the security validation service does not store data, so **dced** maintains no entries for this service.
3. Access (read or write) the actual data for the entries obtained or perform other functions appropriate for the service. For example, if your application reads the host data management service's cell name entry, the API accesses **dced** which may actually read the data from a file. For another example, if your application established a binding to the security validation service, it could validate the Security daemon.
4. Release the resources obtained in step 2.
5. Free the binding established in step 1.

Applications bind to a host service using the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine. The first routine establishes a dced binding to a service on a host specified in a service name, and the second routine establishes a dced binding to a service on a host for which the application already has a binding. Both of the routines return a dced binding handle of type **dced_binding_handle_t**, which is used as an input parameter to all other **dced** API routines.

[1] Applications must establish a binding to each host service used. However, the endpoint mapper service uses a different binding mechanism and API from the other host services. This is due to the fact that the endpoint mapper service already existed within the very large RPC API in earlier versions of DCE, prior to the development of **dced**.

## Host Service Naming in Applications

Applications include a host service name as input to the dced binding routine **dced_binding_create()**. A host service name is a string that may include a host name, or a cell and host name. The following key words in the host service name refer to a specific DCE host service:

| | |
|---|---|
| **hostdata** | The **hostdata** name refers to configuration data of the host data management service. |
| **srvrconf** | The **srvrconf** name refers to the static server configuration portion of the server management service. This refers to the management of a DCE-installed server. |
| **srvrexec** | The **srvrexec** name refers to the dynamic server execution portion of the server management service. This refers to the management of a running DCE-installed server. |
| **secval** | The **secval** name refers to the security validation service. |
| **keytab** | The **keytab** name refers to the private key data of the key table management service. |

The following examples show service names and the locations of the hosts in the namespace:

| | |
|---|---|
| *service* | The host is local, the same as the application's. |
| *service*@**hosts/***host* | The host is in the local namespace. |
| **/.:/hosts/***host***/config/***service* | The complete specification for the previous example where the host is in the local namespace. |
| **/.../***cell***/hosts/***host***/config/***service* | The host is in the global namespace. |

Since the **dced_binding_from_rpc_binding()** routine already knows which host to bind to from an RPC binding input parameter, it uses one of the global variables defined for each service (instead of a string) to specify which **dced** service to use.

## The dced Maintains Entry Lists

One **dced** service's data is very different from another's (for example, server configuration data versus key table data), but you manipulate the data in a similar way. This is because it is a simpler and more efficient design to implement a few API routines that can handle more than one kind of data rather than many routines that do essentially the same thing but on a different service's data. An added benefit is a flexible API that can handle your own application's data and new kinds of DCE data in the future.

To separate the actual data from the API implementation, a **dced** service maintains a list of all data items in an *entry list*. Entry lists contain *entries* that describe the name and location of each item of data, but do not contain the actual data. With this mechanism, **dced** can obtain and manipulate data very efficiently, without concern for the implementation and location of the actual data. It also supports well the model administrators commonly need when accessing data: scan a list, select an item, and use the data.

Figure 1 on page 9 shows the entry lists maintained by **dced**.

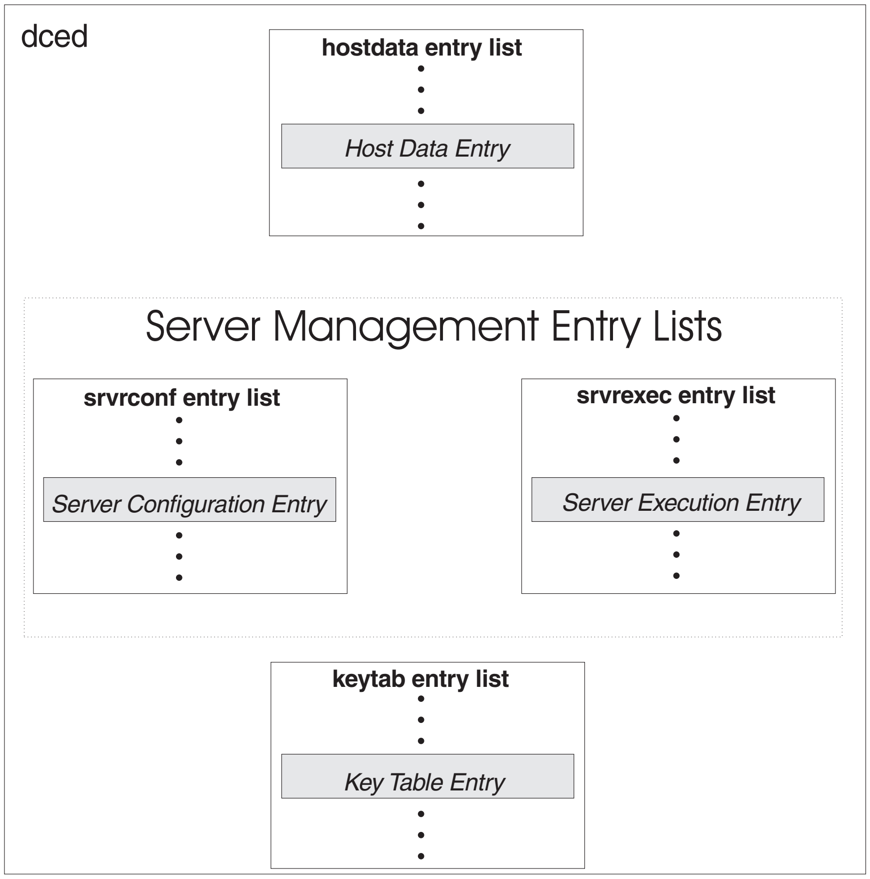


*Figure 1. The dced Entry Lists*

The **dced** maintains entry lists for the **hostdata**, **srvrconf**, **srvrexec**, and **keytab** services. The **secval** service does not need an entry list because it does not maintain any data, but functions are performed to set its state.

There is a special relationship between **srvrconf** and **srvrexec** entries. In order for **dced** to control the start of a server, the server must have a **srvrconf** entry associated with server configuration data. When **dced** starts a server, it generates from the **srvrconf** entry and data a **srvrexec** entry and associates the new entry with the running server's state.

Although an entry can be associated with many different kinds of data items, all entries have the same structure as shown in Figure 2.

Entry UUID, Name, Description, Storage Tag

*Figure 2. Structure of an Entry*

Each entry is a **dced_entry_t** data structure. Each member of this data structure is described as follows:

| | |
|---|---|
| **id** | An entry UUID is necessary to uniquely identify the data item. Some data items have well-known UUIDs (the same UUID for the particular item on all hosts). The data type is **uuid_t**. |
| **name** | Each data item is identified with a name, to which applications refer. The name need only be unique within an entry list, because the entry UUID guarantees the entry's uniqueness. Some item names are well-known and defined in header files. The data type is **dced_string_t**. |
| **description** | This is a human-readable description of the data item. Its data type is **dced_string_t**. |
| **storage_tag** | The storage tag locates the actual data. Each service *knows* how to interpret this tag to find the data. For example, some data is stored in a file, the name of which is contained in the storage tag. Other data is stored in memory and the storage tag contains a pointer to the memory location. The data type is **dced_string_t**. |

## Reading All of a Host Service's Data

Suppose you want to display host service data in an application that has a graphical user interface. The **dcecp** commands may not be adequate to display data for this application. The following example shows how to obtain the entire set of data for each host service:

```
dced_binding_handle_t   dced_bh;
dced_string_t           host_service;
void                    *data_list;
unsigned32              count;
dced_service_type_t     service_type;
error_status_t          status;
 .
 .
 .
while(user_selects(&host_service, &service_type)){   /*application specific*/
    dced_binding_create(host_service,
                    dced_c_binding_syntax_default,
                    &dced_bh,
                    &status);
    if(status == error_status_ok) {
        dced_object_read_all(dced_bh, &count, &data_list, &status);
        if(status == error_status_ok) {
            display(service_type, count, data_list); /*application specific*/
            dced_objects_release(dced_bh, count, data_list, &status);
        }
        dced_binding_free( dced_bh, &status);
    }
}
```

Following is a description of the example:

| | |
|---|---|
| user_selects() | This is an example of an application-specific routine that constructs the complete service name from host and service name information. Data is stored and retrievable for the **hostdata, srvrconf, srvrexec,** and **keytab** services. No data is stored for the **secval** service. |
| **dced_binding_create()** | Output from the **dced_binding_create** routine includes a dced binding handle whose data type is **dced_binding_handle_t**. If an application already has an RPC binding handle to a server on the host desired, it can use the **dced_binding_from_rpc_binding()** routine to bind to **dced** and one of its host services on that host. (Applications also use these routines to bind to the **secval** service to perform other functions.) |

**dced_object_read_all()** Applications use the **dced_object_read_all()** routine to read data for all the objects in an entry list. The output includes the address of an allocated buffer of data and a count of the number of objects the buffer contains. The data type in the buffer depends on the service used.

display() This is an application specific routine that displays the data. Before the data is displayed, it must be interpreted depending on the service. The **hostdata** data is an array of **sec_attr_t** data structures, the **srvrconf** and **srvrexec** data are arrays of **server_t** structures, and the **keytab** data is an array of **dced_key_list_t** structures. The following code fragments show the data type for each service:

```
void display(
dced_service_type_t service_type, /* dced service type */
int                 count,        /* count of the number of data items */
void                *data)        /* obtained from dced_object_read{_all}() */
{
    sec_attr_t               *host_data;
    server_t                *servers;
    dced_key_list_t          *keytab_data;
    .
    .
    .
    switch(service_type) {
    case dced_e_service_type_hostdata:
        host_data = (sec_attr_t *)data;
        . . .
    case dced_e_service_type_srvrconf:
        servers = (server_t *)data;
        . . .
    case dced_e_service_type_srvrexec:
        servers = (server_t *)data;
        . . .
    case dced_e_service_type_keytab:
        keytab_data = (dced_key_list_t *)data;
        . . .
    default:
        /* No other dced service types have data to read. */
        break;
    }
    return;
}
```

**dced_objects_release()** Each call to the **dced_object_read_all()** routine requires a corresponding call to **dced_objects_release()** to release the resources allocated.

**dced_binding_free()** Each call to the **dced_binding_create()** routine requires a corresponding call to **dced_binding_free()** to release the resources for the binding allocated.

## Managing Individual dced Entries

Figure 3 on page 12 shows examples of individual **dced** entries and the locations of associated data. The data item name or its UUID is used to find an entry, and then the storage tag is used to find the data.

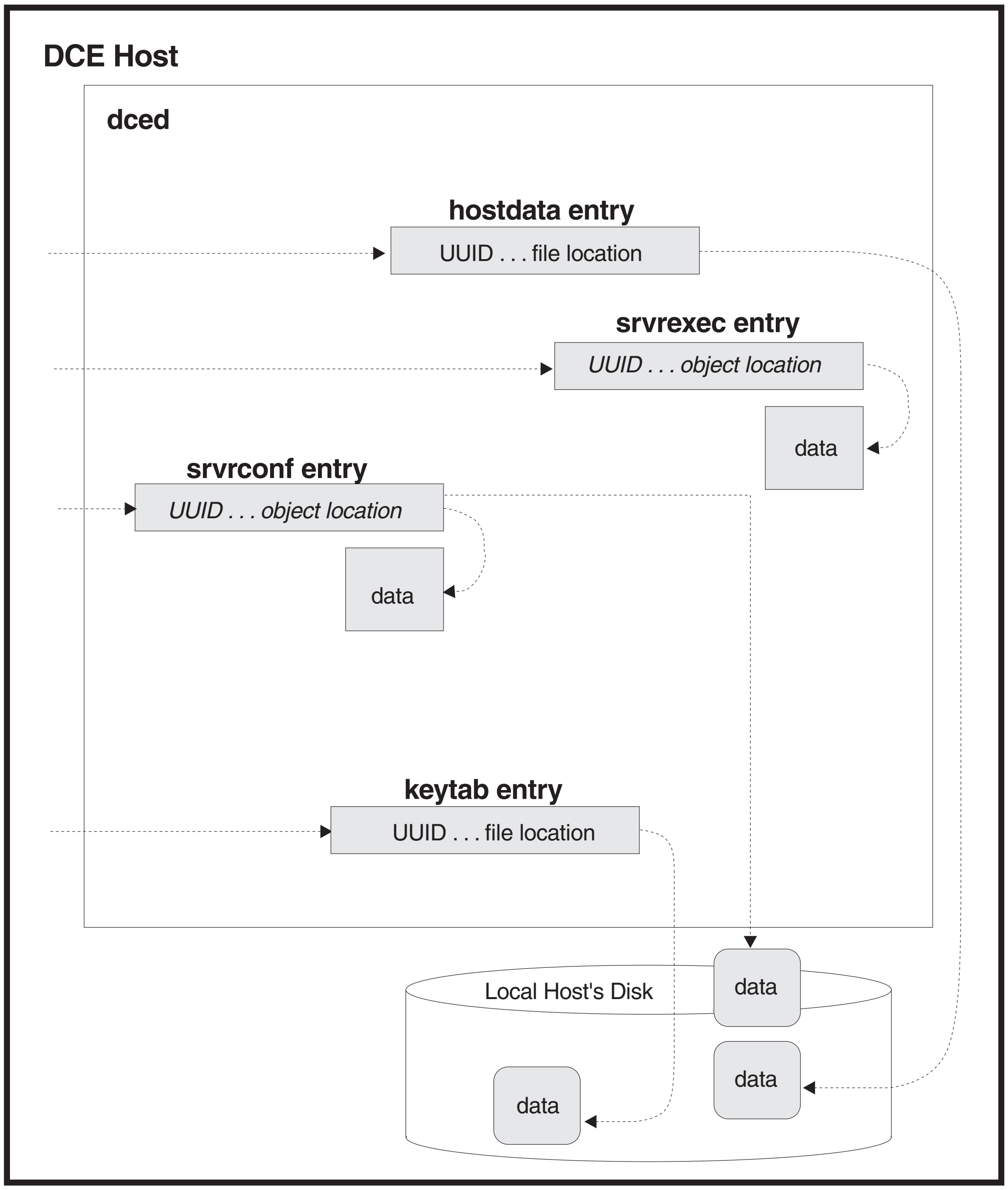


*Figure 3. Accessing Host Data*

The data for each **hostdata** item is stored in a file on disk. The **dced** uses the UUID to find the entry in the hostdata entry list. The entry's storage tag is then used to find the data. For **hostdata**, the tag contains a file name. The data returned for one entry is an array of strings in a **sec_attr_t** structure.

The server management data is stored in memory. The **dced** uses UUIDs (maintained in the entry lists by **dced**) to find an entry. The location of the data in memory is indicated by the storage tag. The data returned for one entry is a structure of server data (**server_t**). All data for the **srvrconf** and **srvrexec** entries are accessed from memory for fast retrieval, but the **srvrconf** data is also stored on disk for use when a host needs to restart DCE.

Each **keytab** entry stores its data in a file on disk. However, like the server management entries, the **keytab** entries use server names and corresponding UUIDs (maintained by **dced**) to identify each entry.

The storage tag contains the name of the key table file. The data returned for one entry is a list of keys of type **dced_key_list_t**.

The following example shows how to obtain and manage individual entries for the **hostdata, srvrconf, srvrexec,** or **keytab** services.

```
handle_t               rpc_bh;
dced_binding_handle_t  dced_bh;
dced_entry_list_t      entries;
unsigned32             i;
dced_service_type_t    service_type;
void                   *data;
error_status_t         status;
.
.
.
dced_binding_from_rpc_binding(service_type, rpc_bh, &dced_bh, &status);
if(status != error_status_ok)
    return;
dced_list_get(dced_bh, &entries, &status);
if(status == error_status_ok) {
    for(i=0; i<entries.count; i++) {
        if( select_entry(entries.list[i].name) ) {/* application specific */
            dced_object_read(dced_bh, &(entries.list[i].id), &data, &status);
            if(status == error_status_ok) {
                display(service_type, 1, &data);  /* application specific */
                dced_objects_release(dced_bh, 1, data, &status);
            }
        }
    }
    dced_list_release(dced_bh, &entries, &status);
}
dced_binding_free(dced_bh, &status);
```

Each routine is described as follows:

**dced_binding_from_rpc_binding()**
The **dced_binding_from_rpc_binding** routine returns a dced binding handle whose data type is **dced_binding_handle_t**. This binding handle is used in all subsequent dced API routines to access the service. The host is determined from the RPC binding handle, `rpc_bh`, and the `service_type` is selected from the following list:

- **dced_e_service_type_hostdata**
- **dced_e_service_type_srvrconf**
- **dced_e_service_type_srvrexec**
- **dced_e_service_type_keytab**

**dced_list_get()**
Applications use the **dced_list_get()** routine to get a service's entire list of names. Using the **dced_list_get()** routine gives your application great flexibility when manipulating entries in an entry list. If you prefer, your application can use the **dced_entry_cursor_initialize()**, **dced_entry_get_next()**, and **dced_entry_cursor_release()** set of routines to obtain individual entries, one at a time.

select_entry()
This is an application specific routine that selects which entry to use based on the entry name.

|  |  |
|---|---|
| **dced_object_read()** | The default attribute for **dced_object_read()** is to return an array of strings. The **hostdata** and **keytab** services have other read routines that allow you to specify binary data. |
| display() | This is an example of an application-specific routine that simply displays the server configuration data read. Depending on the service, a different data structure is used. For the **hostdata** service a **sec_attr_t** is used. For the **srvrconf** and **srvrexec** services **server_t** structures are used. For the **keytab** service a **dced_key_list_t** structure is used. |
| **dced_objects_release()** | After your application is finished with the data read with the **dced_object_read()** routine, free the buffer of data allocated using the **dced_objects_release()** routine. |
| **dced_list_release()** | Each call to the **dced_list_get()** routine requires a corresponding call to **dced_list_release()** to release the resources allocated for the entry list. |
| **dced_binding_free()** | Each call to the **dced_binding_from_rpc_binding()** routine requires a corresponding call to **dced_binding_free()** to release the resources of the allocated binding. |

## Managing Host Data on a Remote Host

Administrators typically use the **dcecp hostdata** object to remotely manage the data of the hostdata service. However, application developers can use the **dced** API for their own management applications or if **dcecp** does not handle a task in the desired way, such as for a browser of host data that uses a graphical user interface.

### Kinds of Host Data Stored

Each hostdata item is stored in a file and **dced** has a UUID associated with each. On z/OS DCE, the standard data items include the following well-known names:

|  |  |
|---|---|
| **cell_name** | The name of the cell to which your host belongs is stored. |
| **cell_aliases** | When the cell name changes, the old names are designated as cell aliases. |
| **dce_cf.db** | The DCE configuration data file is stored. |
| **host_name** | The host name is stored. |
| **pe_site** | The location of the Security server is stored. |
| **post_processors** | The **post_processors** file contains UUID-program pairs for which the UUIDs represent other hostdata items. If changes occur to an associated hostdata item, the system runs the program. |

In addition to the well-known hostdata items, applications can also add their own. (DCE implementations other than z/OS DCE may also define additional items.) The well-known hostdata items have well-known UUIDs defined in the file **/usr/include/dce/dced_data.h**, but you can use the **dced_inq_uuid()** routine to obtain any UUID associated with any name known to **dced**.

See the *z/OS DCE Administration Guide* for additional information on managing host data.

## Adding New Host Data

In addition to modifying existing host data, you can add your own data by using the host data API. For example, suppose you want to add a printer to a host, and make the configuration file part of that host's **dced** data. The following example shows how to do this:

```
dced_binding_handle_t    dced_bh;
error_status_t           status;
dced_entry_t             entry;
dced_attr_list_t         data;
int                      num_attr, str_size;
sec_attr_enc_str_array_t *attr_array;
 .
 .
 .
dced_binding_create(dced_c_service_hostdata,
                dced_c_binding_syntax_default,
                &dced_bh,
                &status);
/*Create Entry Data */
uuid_create(&(entry.id), &status);
entry.name = (dced_string_t)("NEWERprinter");
entry.description = (dced_string_t)("Configuration for a new printer.");
entry.storage_tag = (dced_string_t)("/etc/NEWprinter");

/* Create the Attributes, one for this example */
data.count = 1;
num_attr = 1;
data.list  = (sec_attr_t *)malloc( data.count * sizeof(sec_attr_t) );
(data.list)->attr_id = dced_g_uuid_fileattr;
(data.list)->attr_value.attr_encoding = sec_attr_enc_printstring_array;
str_size = sizeof(sec_attr_enc_str_array_t) +
                      num_attr * sizeof(sec_attr_enc_printstring_p_t);
attr_array = (sec_attr_enc_str_array_t *)malloc(str_size);
(data.list)->attr_value.tagged_union.string_array = attr_array;
attr_array->num_strings = num_attr;
attr_array->strings[0] = (dced_string_t)("New printer configuration data");

dced_hostdata_create(dced_bh, &entry, &data, &status);
dced_binding_free( dced_bh, &status);
```

Following is a description of the example:

| | |
|---|---|
| **dced_binding_create()** | This routine creates a dced binding to a **dced** service. The binding handle created is used in all subsequent calls to appropriate dced API routines. By using the **dced_c_server_hostdata** value for the first parameter, we are using the hostdata service on the local host. |
| Create Entry Data | Prior to creating a hostdata entry, we have to set its values. These include the name and UUID that **dced** will use to identify the new data, a description of the entry, and a file name with the full path where the actual data will reside. |
| Create the Attributes | The data stored is of type **sec_attr_t**. This data type is a very flexible one which can store many different kinds of data. In this example we set the file to have one attribute, printable string information. This example has only one string of data. You can also establish binary data for the file. |
| **dced_hostdata_create()** | This routine takes the binding handle, entry, and new data as input, creates the file with the new data, and returns a status code. If the printer configuration file already exists on the host, but you want to make |

it accessible to **dced**, use the **dce_entry_add()** routine instead of **dced_hostdata_create()**.

**dced_binding_free()** Each call to the **dced_binding_create()** routine requires a corresponding call to **dced_binding_free()** to release the binding resources allocated.

Use the **dced_hostdata_delete()** routine to delete application-specific hostdata items and their entries. For example, the printer installed in the above example is easily removed with this routine. If you are only taking the printer out of service for a short time, use the **dced_entry_remove()** routine to remove the **dced** entry but not the data file itself. When the printer is later ready again, use the **dced_entry_add()** routine to re-install it.

Do not delete the well-known hostdata items or remove their entries.

## Modifying Host Data

Changing host data can not only change the way the host works but it affects other files and processes on the host. Therefore, care should be taken when changing host data. Deleting the well-known hostdata entries can cause even more serious operational problems for the host.

The current as well as earlier versions of DCE provide configuration routines that use a **dce_cf.db** file for data. When host data changes, **dced** also makes the appropriate changes to this file so that the **dce_cf*** routines continue to work correctly. This is one reason the hostdata items are established as well-known names with well-known UUIDs so that **dced** knows which values to monitor.

Management applications use the **dced_hostdata_read()** routine to obtain the data for an entry referred to by an entry UUID. To modify an entry's actual data, applications use the **dced_hostdata_write()** routine. This routine replaces the old data with the new data for the host data entry represented by the entry UUID. The host data entry must already exist because this routine will not create it. Use the **dced_hostdata_create()** routine to create new host data entries.

## Running Programs Automatically When Host Data Changes

The following example shows how to use the **post_processors** feature of the well-known hostdata to cause **dced** to automatically run a program if another hostdata entry changes. In this example, the post_processors file is read, and data is added for the **NEWERprinter** hostdata entry created in an earlier example. The data is placed in a **dced_attr_list_t** structure and written back to the **post_processors** hostdata entry.

```
dced_binding_handle_t dced_bh;
uuid_t                entry_uuid;
sec_attr_t            *data_ptr;
error_status_t        status;
int                   i, num_strings, str_size;
sec_attr_enc_str_array_t *attr_array;
unsigned_char_t       *string_uuid, temp_string[200];
dced_attr_list_t      attr_list;

dced_binding_create(dced_c_service_hostdata,
                dced_c_binding_syntax_default,
                &dced_bh,
                &status);
dced_hostdata_read(dced_bh,
                   &dced_g_uuid_hostdata_post_proc,
                   &dced_g_uuid_fileattr,
                   &data_ptr,
                   &status);
```

```
/* Create New Array and Copy Old Data into it */
num_strings = data_ptr->attr_value.tagged_union.string_array->num_strings + 1;
str_size = sizeof(sec_attr_enc_str_array_t) +
                      num_strings * sizeof(sec_attr_enc_printstring_p_t);
attr_array = (sec_attr_enc_str_array_t *)malloc(str_size);
attr_array->num_strings = num_strings;
for(i=0; i<(num_strings-1); i++) {
    attr_array->strings[i] =
        data_ptr->attr_value.tagged_union.string_array->strings[i];
}

dced_inq_id(dced_bh, "NEWERprinter", &entry_uuid, &status);

uuid_to_string(&entry_uuid, &string_uuid, &status);
sprintf(temp_string, "%s %s", string_uuid, "/path/and/program/to/run");
attr_array->strings[num_strings-1] = (dced_string_t)(temp_string);
data_ptr->attr_value.tagged_union.string_array = attr_array;

attr_list.count = 1;
attr_list.list = (sec_attr_t *)malloc(attr_list.count * sizeof(sec_attr_t));
attr_list.list = data_ptr;
dced_hostdata_write(dced_bh,
                    &dced_g_uuid_hostdata_post_proc,
                    &attr_list,
                    &status);

dced_objects_release(dced_bh, 1, (void*)(data_ptr), &status);
dced_binding_free(dced_bh, &status);
```

The example is described as follows:

**dced_binding_create()** This routine creates a dced binding to the hostdata service on a specified host. The binding handle created is used in all subsequent calls to appropriate dced API routines. The **dced_c_service_hostdata** argument is a constant string that is the well-known name of the hostdata service. When this string is used by itself, it refers to the service on the local host.

**dced_hostdata_read()** This routine reads the hostdata item referred to by the entry UUID. In this example, the global variable **dced_g_uuid_hostdata_post_proc** represents the UUID for the well-known post_processors file. The second parameter specifies an attribute for the data. Attributes describe how the data is to be interpreted. In this example we know the data to be read is plain text so we use the global variable **dced_g_uuid_fileattr** to specify plain text rather than binary data (**dced_g_uuid_binfileattr**).

**Create New Array** The next few lines copy the existing array of print strings into a new array that has additional space allocated for the new data.

**dced_inq_id()** This routine acquires the UUID **dced** maintains for a known entry name. In this example, we need the UUID for the **NEWERprinter** hostdata entry, so it can be included in the data stored back in the post_processors file.

**uuid_to_string()** This routine returns the string representation of a UUID. Each line in the post_processors file contains a string UUID and a program name for **dced** to run if the hostdata entry referred to by the UUID changes. The next few lines create a new string containing the string UUID and a program name, adds the new string to the new array, and reassigns the new array to the old data pointer.

| | |
|---|---|
| **dced_hostdata_write()** | Since hostdata could have more than one attribute associated with each entry, the data must be inserted in an attribute list data structure before the **dced_hostdata_write()** routine is called. In the case of the well-known post_processor hostdata object, the attribute is for a plain text file. The **dced_hostdata_write()** routine replaces the old data with the new data for the hostdata entry represented by the entry UUID. |
| **dced_objects_release()** | Each call to the **dced_hostdata_read()** routine requires a corresponding call to **dced_objects_release()** to release the resources allocated. |
| **dced_binding_free()** | Each call the **dced_binding_create()** routine requires a corresponding call to **dced_binding_free()** to release the resources allocated. |

The post_processors data for this **dced** now contains an additional string with a UUID and program name. If the hostdata item represented by the UUID for **NEWERprinter** is changed, **dced** automatically runs the program.

**Note:** In z/OS UNIX System Services if the post processor program is a shell script, the first two characters of the file must be **# !**. They should be entered using the same code page that DCEKERN will be using at the time the post-processor is executed.

If DCEKERN requires a code page other than IBM-1047, use the z/OS **iconv** command to convert **/opt/dcelocal/bin/dcecf_postproc** to the new code page before starting DCEKERN. As initially installed, **dcecf_postproc** (a post-processor shell script) is in the IBM-1047 code page. See the *z/OS UNIX System Services Command Reference*, SA22-7802, for information on the **iconv** command.

The shell (**/bin/sh**) will be invoked to execute the script. **stdin**, **stdout**, and **stderr** will not be open and the post-processor program must open them as necessary, for example, by using redirection with a shell script to route output to an HFS file.

## Controlling Servers Remotely

Both applications developers and system administrators may want servers to have certain support services and control functionality. For example, servers may need mechanisms to store operational data, and they may need to start or stop in various ways. The **dced** program provides these support and control mechanisms for servers.

Servers are typically configured by an administrator using the **dcecp server** object in a script after the server is installed on the host. In addition to configuring the server, this script would commonly include other tasks like create an account and assign a principal name for the server, modify the access control lists (ACLs) and key table files (keytabs) to control access to the server and its resources, and export the server binding information to the Cell Directory Service (CDS) so that clients can find a server that will start dynamically later.

After a server is configured, whether it runs as a persistent daemon or an on-demand (dynamic) process, administrators would again use **dcecp** if they need to control or modify its behavior. Although server management is typically an administrator's task, you may want a management application to perform these tasks, including the following:

- Configure a server to describe how it can be invoked
- Start a server based on configuration data
- Stop a running server
- Disable a specific service provided by a running server
- Enable a specific service for a running server

- Modify a server's configuration
- Delete a server's configuration, effectively removing the server from **dced**'s control

## Two States of Server Management: Configuration and Execution

If all servers ran as persistent processes, **dced** could maintain data about each server in a single (albeit complex) data structure. However, due to the fact that some servers may run on-demand, it is a more flexible design to have two sets of data: one that describes the default configuration to start the server, and one that describes the executing (running) server. Earlier in this chapter when we described **dced** service naming, we defined **srvrconf** and **srvrexec** objects to name the two portions of the server management service.

Table 1 lists the routines applications can use to control servers. It also shows the valid object names to use when establishing a dced binding prior to using the routine.

*Table 1. API Routines for Remote Server Management*

| API Routine | Service Name for Binding |
|---|---|
| **dced_server_create()** | **srvrconf** |
| **dced_server_start()** | **srvrconf** |
| **dced_server_disable_if()** | **srvrexec** |
| **dced_server_enable_if()** | **srvrexec** |
| **dced_server_stop()** | **srvrexec** |
| **dced_object_read()** | **srvrexec** or **srvrconf** |
| **dced_object_read_all()** | **srvrexec** or **srvrconf** |
| **dced_server_modify_attributes()** | **srvrconf** |
| **dced_server_delete()** | **srvrconf** |

## Configuring Servers

Although administrators commonly use **dcecp** to configure servers remotely, management applications can use **dced** API routines to configure a new server remotely by creating server configuration data, changing a remote server's configuration, and deleting a server's configuration data.

**Configuring a New DCE Server:** Management applications use the **dced_server_create()** routine to add a new server to a host. After a server is configured, it can be remotely controlled by modifying its configuration attributes, starting and stopping it, enabling or disabling the RPC interfaces it supports, and deleting its configuration.

Configuring the server involves describing the server for DCE by allocating and filling in a **server_t** data structure, as shown in the following example. Note that not all **server_t** fields are assigned values in the following example:

```
int                   i;
dced_binding_handle_t dced_bh;
server_t              conf, exec;
dced_string_t         server_name;
uuid_t                srvrconf_id, srvrexec_id;
dced_attr_list_t      attr_list;
error_status_t        status;
static service_t      nil_service;
.
```

```
.
.
dced_binding_create("srvrconf@hosts/somehost",
                    dced_c_binding_syntax_default,
                    &dced_bh,
                    &status);
dced_inq_id(dced_bh, server_name, &srvrconf_id, &status);
if(status == error_status_ok) {
    puts("Configuration already exists for this server.");
    dced_binding_free(dced_bh, &status);
    return;
}
/* __________setup a server_t structure______________*/
uuid_create(&(conf.id), &status);
conf.name           = server_name;
conf.entryname      = (dced_string_t)"/.:/greeter";
conf.services.count = 1;

/* ___service_t structures represent each interface supported ___*/
conf.services.list =
    (service_t *)malloc(conf.services.count * sizeof(service_t));
for(i=0; i<conf.services.count; i++) {
    rpc_if_inq_id(greetif_v1_0_c_ifspec,
                  &(conf.services.list[i].ifspec),
                  &status);
    conf.services.list[i] = nil_service;
    conf.services.list[i].ifname     = (dced_string_t)"greet";
    conf.services.list[i].annotation = (dced_string_t)"The greet application";
    conf.services.list[i].flags      = 0;
}

/* _____________server_fixedattr_t structure_____________*/
conf.fixed.startupflags =
    server_c_startup_explicit | server_c_startup_on_failure;
conf.fixed.flags = 0;
conf.fixed.program = (dced_string_t)"/server/path/and/program/name";

dced_server_create(dced_bh, &conf, &status);
dced_binding_free(dced_bh, &status);
```

**dced_binding_create()** To configure a server, the application must first create a dced binding to the **srvrconf** portion of the server management service on a specified host. The binding handle created is used in all subsequent calls to appropriate dced API routines.

**dced_inq_id()** This routine returns the UUID that **dced** associates with the name input. Each configured server has an associated UUID used by **dced** to identify it. This example does not try to create a configuration for a server that already exists.

Setup a **server_t** Structure for the Server
The **server_t** structure contains all the information DCE uses to specify a server.

Setup **service_t** Structures for each Interface
Each service that the server supports is represented by a **service_t** data structure which contains the interface specification among other things. In this example, the client stub for the interface was compiled with the program so that the interface specification (`greetif_v1_0_c_ifspec`) could be obtained without building the structure from scratch.

Setup a **server_fixedattr_t** Structure
Other fixed attributes required for all servers describe how the server can start, the program path and name for the server so that **dced** knows which program to start, and the program's arguments, among other things.

**Note:** In z/OS UNIX System Services if the program is a shell script, the first two characters of the file must be **# !**. They should be entered using the same code page that DCEKERN will be using at the time the program is executed. The shell (**/bin/sh**) will be invoked to execute the script. **stdin**, **stdout**, and **stderr** will not be open and the post-processor program must open them as necessary, for example, by using redirection with a shell script to route output to an HFS file.

**dced_server_create()** This routine uses the filled-in **server_t** structure to create a **srvrconf** entry for **dced**. The data is stored in memory for quick access whenever the server is started.

**dced_binding_free()** Each call to the **dced_binding_create()** routine requires a corresponding call to **dced_binding_free()** to release the binding resources allocated.

**Modifying a Server's Configuration Attributes:** The data for configuring servers includes arrays of attributes. For flexibility, the **dced** is implemented using the extensible and dynamic data structures developed for the DCE security registry attributes. This extended registry attribute (ERA) schema gives vendors the flexibility to modify the attributes appropriate for configuring servers on various systems. The use and modification of these data structures are described in Chapter 29, “The Extended Attribute Application Program Interfaces” on page 465.

Applications commonly use **dced_server_modify_attributes()** after the **dced_server_create()** routine to change the default configuration attributes (the **attributes** field of a **server_t** structure) for a remote server. A **dced_attr_list_t** data structure is input which contains an array of **sec_attr_t** data structures and a count of the number in the array.

**Deleting a DCE Server:** Management applications use **dced_server_delete()** to delete a server's configuration data and entry in its hosts **dced**. Although this does not delete the actual server program from the host, it removes it from DCE control.

## Starting and Stopping Servers

Servers typically run as persistent processes or are started on demand when a client makes a remote procedure call to it. Management applications can start remote servers using the **dced_server_start()** routine. This is a **srvrconf** routine that takes as input server configuration data in the form of an attribute list.

Once a server has started, it tends to remain running until an administrator or management application stops it, but some applications may stop themselves if, for example, they do not detect activity within a specified time. To stop remote servers, applications can use the **dced_server_stop()** routine.

The following example shows how an application starts or stops a server:

```
dced_binding_handle_t dced_bh, conf_bh, exec_bh;
server_t              conf, exec;
dced_string_t         server_name;
uuid_t                srvrconf_id, srvrexec_id;
error_status_t        status;
.
.
.
/* Toggle the Starting or Stopping of a Server */
dced_binding_create("srvrconf@hosts/somehost",
                    dced_c_binding_syntax_default,
                    &conf_bh,
                    &status);
```

```
    dced_binding_create("srvrexec@hosts/somehost",
                        dced_c_binding_syntax_default,
                        &exec_bh,
                        &status);
    dced_inq_id(exec_bh, server_name, &srvrexec_id, &status);
    if(status != error_status_ok) {
        puts("Server is NOT running.");
        dced_inq_id(conf_bh, server_name, &srvrconf_id, &status);
        dced_server_start(conf_bh, &srvrconf_id, NULL, &srvrexec_id, &status);
    }
    else {
        puts("Server is RUNNING.");
        dced_server_stop(exec_bh, &srvrexec_id, srvrexec_stop_soft, &status);
    }
    dced_binding_free(conf_bh, &status);
    dced_binding_free(exec_bh, &status);
```

|  |  |
|---|---|
| **dced_binding_create()** | These routines create dced bindings to the **srvrconf** and **srvrexec** portions of the server management service on a specified host. The binding handles created are used in all subsequent calls to appropriate dced API routines. |
| **dced_inq_id()** | This routine returns the UUID that **dced** associates with the name input. Each name used to identify an object of each service has a UUID. If **dced** maintains a UUID for a **srvrexec** object, the server is running. However, it is possible that the server is in an in-between state as it is starting up or shutting down. For a more robust check as to whether the server is running, use the **dced_object_read()** routine to read the **server_t** structure for the **srvrexec** object. If the `exec_data.tagged_union.running_data.instance` UUID is the same as the **srvrconf** UUID (`srvrconf_id`), the server is running. |
| **dced_server_start()** | This routine starts the server via **dced**. The **srvrconf** binding handle and UUID are input. For special server configurations, you can start a server with a specific list of attributes, but a value of NULL in the third parameter uses the attributes of the server configuration data. You can input a **srvrexec** UUID for **dced** to use, or allow it to generate one for you. |
| **dced_server_stop()** | This routine stops a running server identified by its **srvrexec** UUID. |
| **dced_binding_free()** | Each call to the **dced_binding_create()** routine requires a corresponding call to **dced_binding_free()** to release the binding resources allocated. |

## Enabling and Disabling Services of a Server

Most servers have all their services enabled to process all requests. However, a server may need to enable or disable services to synchronize them, for example. For another example, an administrator (or management application) may need to disable or enable services to perform orderly startup or shutdown of a server. Each service provided by a server is implemented as a set of procedures. DCE uses an interface definition to define a service and its procedures, and application code refers to the interface when controlling the service.

When a server starts, it initializes itself by registering with the RPC runtime and the **dced** on its host using the **dce_server_register()** routine. This enables all services (interfaces) that the server can support.

**Note:** The **dce_server_disable_if()** and **dce_server_enable_if()** routines are not supported when targeted at DCE hosts running on z/OS. If you wish to have clients that already know about the server and service work, but wish to prohibit any new clients from finding the server and service, you can use **rpc_mgmt_ep_unregister()** to remove from the endpoint map the server address information with respect to the service. This routine does not affect the RPC runtime.

## Validating the Security Server

The security validation service (**secval**) has the following major functions:

- It maintains a login context for the host's self identity which includes periodic changes to the host's key (password).
- It validates and certifies to applications, usually login programs, that the DCE Security Daemon (**secd**) is legitimate.

Clients (including remote clients, local servers, host logins, and administrators) all need the security validation service to make sure the DCE Security Daemon (**secd**) being used by the host is legitimate. The security validation service establishes the link in a trust chain between applications and **secd**, so that applications can trust the DCE security mechanism.

An application can trust its host's security validation service because they are on the same host, but an application has no way to convince itself that **secd**, presumably on another host, is genuine. However, if the application trusts another principal (in this case the security validation service), which in turn trusts **secd**, then the trust chain extends from the application to **secd**.

Typically, a login program accesses the security validation service when it uses the Security Service's Login API, described in Chapter 30, “The Login Application Program Interface” on page 495. Administrators access the **secval** service by using **dcecp**'s **secval** object. However, suppose you are writing a security monitoring application to watch for and respond to security attacks. After the application binds to the **secval** service, it can call the **dced_secval_validate()** routine to verify that the **secd** is legitimate.

Applications can also use the **dced_secval_start()** and **dced_secval_stop()** routines to start and stop the security validation service on a given host.

For example, during configuration of a host, the **dced** program can start with or without the security validation service. Later when security is configured, a management application can start **secval** using the **dced_secval_start()** routine. For another example, suppose our security monitoring application mentioned earlier suspects an attack. The application can call **dced_secval_stop()** to stop the security validation service without stopping the entire **dced**. This makes the login environment more restrictive.

## Managing Server Key Tables

Keys for servers are analogous to passwords for human users. Keys also play a major role in authenticated remote procedure calls. Keys have some similarities with passwords. For example, server keys and user passwords have to follow the same change policy (or a more stringent one) for a given host or cell. This means that just as a user has to periodically come up with a new password, a server has to periodically generate a new key. It is easy to see that a human user protects a password by memorizing it. But a server *memorizes* a key by storing it in a file called a key table.

It is more complex for a server to change keys than it is for a human user to change a password. For example, a human user needs to only remember the latest password, but a server may need to maintain a history of its keys using version numbers, so that currently active clients do not have difficulty completing a remote procedure call. When a client prepares to make authenticated remote procedure calls, it obtains a ticket to talk with the server. (The security registry of the authentication service encrypts this ticket using the server's key, and later the server decrypts the ticket when it receives the remote procedure call.) Timing can become an issue when a client makes a remote procedure call, because tickets have a limited lifetime before they expire, and servers must also change their keys on a regular basis. Assuming the client possesses a valid ticket, suppose that by the time the client makes the call, the server has generated a new key. If a server maintains versions of its keys, the client can still complete the call. Authentication is described in detail in Chapter 24, “Authentication” on page 413.

A key table usually contains keys stored by one server, and must be located on the same host as that server. However, a key table can hold keys for a set of related servers, as long as all the servers reside on the same host. Servers usually maintain their own keys, and Chapter 31, “The Key Management Application Program Interface” on page 501 describes the API they use. Administrators can remotely manage key tables and the keys in the tables by using **dcecp**'s **keytab** object. This section describes the API routines that management applications can use to manage the key tables and keys of other servers on the network.

Suppose you discover that a server or an entire host's security has been compromised. Applications can use the **dced_keytab_change_key()** routine to change a key table's key. The following example shows how to reset the key for all key tables on a specified host:

```
dced_binding_handle_t   dced_bh;
dced_entry_list_t       entries;
unsigned32              i;
error_status_t          status;
dced_key_t              key;

dced_binding_create("keytab@hosts/somehost",
                     dced_c_binding_syntax_default,
                     &dced_bh,
                     &status);

dced_binding_set_auth_info(dced_bh,
                           rpc_c_protect_level_default,
                           rpc_c_authn_default,
                           NULL,
                           rpc_c_authz_dce,
                           &status);

dced_list_get(dced_bh, &entries, &status);

for(i=0; i<entries.count; i++) {
    generate_new_key(&key); /* application specific */
    dced_keytab_change_key(dced_bh, &entries.list[i].id, &key, &status);
}
dced_list_release(dced_bh, &entries, &status);
dced_binding_free( dced_bh, &status);
```

Each routine used in this example is described here:

**dced_binding_create()** This routine creates a dced binding to a **dced** service on a specified host. The binding handle created is used in all subsequent calls to appropriate dced API routines. The **keytab** portion of the first argument represents the well-known name of the keytab service. When this string is used by itself, it refers to the service on the local host.

**dced_binding_set_auth_info()** Accessing keytab data requires authenticated remote procedure calls. The **dced_binding_set_auth_info()** routine sets authentication for the dced binding handle, `dced_bh`.

**dced_list_get()** Applications use the **dced_list_get()** routine to get a service's entire list of names.

|  |  |
|---|---|
| **generate_new_key()** | This application-specific routine generates the new key, and fills in a **dced_key_t** data structure. This routine could use the **sec_key_mgmt_gen_rand_key()** routine to randomly generate a new key. |
| **dced_keytab_change_key()** | The **dced_keytab_change_key()** routine tries to change principal's key in the Security Service's registry first. If that is successful, it changes the key in the key table. |
| **dced_list_release()** | Each call to the **dced_list_get()** routine requires a corresponding call to **dced_list_release()** to release the resources allocated for the entry list. |
| **dced_binding_free()** | Each call to the **dced_binding_create()** routine requires a corresponding call to **dced_binding_free()** to release the resources allocated for a dced binding handle. |

For more detailed key table management, applications can peruse a key table's list of keys by using the **dced_keytab_initialize_cursor()**, **dced_keytab_get_next_key()**, and **dced_keytab_release_cursor()** routines. Reading key table data remotely presents a greater security risk because data is sent over the network. For remote access these routines actually get all the keys during one remote procedure call to be more efficient and to minimize the time keys are being sent over the network.

Earlier in this section we described how to change the key of a key table with the **dced_keytab_change_key()** routine. The key table management service also provides the routines **dced_keytab_add_key()** and **dced_keytab_remove_key()** to control key modification in even greater detail.

Finally, you can create a new key table using **dced_keytab_create()** or delete an existing key table using **dced_keytab_delete()**.

# Chapter 3. The DCE Backing Store

This chapter describes the backing store library that DCE provides for the convenience of programmers who are writing DCE servers. A backing store is a persistent database or persistent object store from which typed data can be stored and retrieved by a key.

**Note:** Sometimes the backing store is called a database. For instance, the associated IDL file is **dce/database.idl**, and the name of the backing store routines begin with **dce_db_**. The backing store is, however, not a full-fledged database in the conventional sense, and it has no support for SQL or for any other query system.

Servers generally need to manage several objects. Good design often requires that the state of the objects be maintained over sequential instances of a particular server. For example, the ACLs used by a server should not need to be recalculated each time the system is rebooted. The backing store interface provides a way to store, into a file, any data that can be described with IDL, so that it can persist across instances of software that run from time to time. For example, the ACL library uses the backing store library. The backing store routines can be used in servers, in clients or in stand-alone programs that do not involve remote procedure calls (RPC). Backing store data should not be used for sharing data between processes.

## Data in a Backing Store

The backing store interface provides the application programmer with the capability for tagged storage and retrieval of typed data. The tag (or retrieval key) can be either a UUID or a standard C string. For a specific backing store, the data type must be specified at compile time, and is established through the IDL Encoding Services. Each backing store can contain only a single data type.

Each data item (which also may called a data object, or a data record) consists of the data stored in a single call to a storage routine. The storage routines are **dce_db_store()**, **dce_db_store_by_name()**, and **dce_db_store_by_uuid()**. Optionally, data items may have standard headers. If a backing store has been created to use headers, then every data item has the header.

A program can have more than one backing store open at the same time.

## Using A Backing Store

Although the backing store library is a generalized service, you are encouraged to use it in a particular, standardized way. You should use the header and the recommended IDL interface format that are described in the following sections. Standardized use will ease the transition to later developments in DCE.

## Header for Data

An optional standard header is available for data objects or items in the backing store. If it is employed, then the backing store library automatically maintains the **created**, **modified**, and **modified_count** fields, as shown in the following IDL description taken from the **dce/database.idl** file:

```
/* This is the standard header for each "object" in the database. */

typedef struct dce_db_dataheader_s_t {
    uuid_t            uuid;
    uuid_t            owner_id;
    uuid_t              group_id;
    uuid_t              acl_uuid;
    uuid_t              def_object_acl;
    uuid_t              def_container_acl;
    unsigned32          ref_count;
    /* The following fields are updated by the library */
    utc_t               created;
    utc_t               modified;
    unsigned32          modified_count;
} dce_db_dataheader_t;

typedef enum {
    dce_db_header_std,
    dce_db_header_acl_uuid,
    dce_db_header_none
} dce_db_header_type_t;

typedef union switch (dce_db_header_type_t type) tagged_union {
    case dce_db_header_none:            /* none */ ;
    case dce_db_header_std:             dce_db_dataheader_t h;
    case dce_db_header_acl_uuid:        uuid_t  acl_uuid;
} dce_db_header_t;

void dce_db_header_convert(
    [in]        handle_t                h,
    [in,out]    dce_db_header_t         *data,
    [out]       error_status_t          *st
);
```

The **acl_uuid** field is intended for use as a UUID retrieval key in a server's ACL database.

## The User Interface

The recommended, standardized backing store IDL interface for a server looks like the following, where *XXX* is the server name:

```
interface XXX_convert
{
    import "dce/database.idl"

    typedef XXX_data_s_t {
        dce_db_header_t  header;   /* Header must be first! */
        /* (server-specific data goes here) */
    } XXX_data_t;

    void XXX_data_convert(
        [in]        handle_t        h,
        [in, out]   XXX_data_t      *data
        [out]       error_status_t  *st
    );
}
```

It should be compiled with the following ACF, that instructs the **idl** compiler to write the data conversion routine into the *XXX*_**cstub.c** file:

```
interface XXX_convert
{
    [encode, decode] XXX_data_convert([comm_status] st);
}
```

## The IDL Encoding Services

When a Remote Procedure Call (RPC) sends data between a client and a server, it serializes the user's data structures by using the IDL Encoding Services (ES), described in “Creating Portable Data Using the IDL Encoding Services” on page 211.

## Encoding and Decoding in the Backing Store

The backing store uses this same serialization scheme for encoding and decoding, informally called *pickling*, when storing data structures to disk. The IDL compiler, **idl**, writes the routine that encodes and decodes the data. This routine is passed to **dce_db_open()**, remembered in the handle, and used by the store and fetch routines:

- **dce_db_fetch()**
- **dce_db_fetch_by_name()**
- **dce_db_fetch_by_uuid()**
- **dce_db_header_fetch()**
- **dce_db_store()**
- **dce_db_store_by_name()**
- **dce_db_store_by_uuid()**

## Conformant Arrays Not Allowed

You can not use conformant arrays in objects stored to a backing store. This is because the IDL-generated code that encodes (pickles) the structure has no way to predict or detect the size of the array. When the object is fetched, there will likely be insufficient space provided for the structure, and the array's data will destroy whatever is in memory after the structure.

To illustrate the problem more clearly, here is an example. An IDL file has a conformant array, **na**, as an object in a struct:

```
typedef struct {
    unsigned32 length;
    [size_is(length)]
    unsigned32 numbers[];
} num_array_t
typedef struct {
    char          *name;
    num_array_t   na;
} my_type_t;
```

The **idl** compiler turns the IDL specification into the following **.h** file contents:

```
typedef struct  {
    unsigned32 length;
    unsigned32 numbers[1];
} num_array_t
typedef struct {
    idl_char       *name;
    num_array_t     na;
} my_type_t;
```

When the object is fetched, and the array length is greater than the 1 (one) assumed in the **.h** file, the decoding operation destroys whatever follows **my_struct** in memory:

```
my_type_t  my_struct;
dce_db_fetch(dbh, key, &my_struct, &st);
```

The correct method is to use a pointer to the array, not the array itself, in the IDL file. For example:

```
typedef struct {
    char          *name;
    num_array_t   *na;
} my_type_t;
```

## The Backing Store Routines

Many of the backing store routines appear in three versions: plain, by name, and by UUID. The plain version will work with backing stores that were created to be indexed either by name, or by UUID, while the restricted versions accept only the matching type. It is advantageous to use the restricted versions when they are appropriate, because they provide type checking by the compiler, as well as visual clarity of purpose.

The backing store operations described in the following sections are supported.

### Opening a Backing Store

The **dce_db_open()** routine creates a new backing store or opens an existing one. The backing store is identified by a file name. There are flags to permit the following choices:

- Create a new backing store, or open an existing one.
- Create a new backing store indexed by name, or indexed by UUID. (The choice depends upon the server's purpose.) This index is called the backing store key.
- Open an existing backing store read/write, or read-only.
- Use the standard header, or not.

Every backing store is created with one of the two possible index schemes, by name or by UUID, and you cannot subsequently open it for use with the other scheme. Also, once a backing store has been created with (or without) standard headers, you cannot subsequently open it the other way.

The routine returns a handle by which subsequent operations identify the backing store.

The following conventions for file names are recommended:

| | |
|---|---|
| *xxx***.acl** | ACL storage. |
| *xxx***.db** | Backing store file name. |

### Closing a Backing Store

The **dce_db_close()** routine frees the handle. It closes any open files and releases all other resources associated with the backing store.

## Storing or Retrieving Data

The following routines store data into a backing store:

| | |
|---|---|
| **dce_db_store()** | This routine can store into a backing store that is indexed by name or by UUID. The key's type must match the flag that was used in **dce_db_open()**. |
| **dce_db_store_by_name()** | This routine can store only into a backing store that is indexed by name. |
| **dce_db_store_by_uuid()** | This routine can store only into a backing store that is indexed by UUID. |

To retrieve data from a backing store, use the appropriate one of the following routines:

| | |
|---|---|
| **dce_db_fetch()** | This routine can retrieve data from a backing store that is indexed by name or by UUID. The key's type must match the flag that was used in **dce_db_open()**. |
| **dce_db_fetch_by_name()** | This routine can retrieve data only from a backing store that is indexed by name. |
| **dce_db_fetch_by_uuid()** | This routine can retrieve data only from a backing store that is indexed by UUID. |

When storing or retrieving data, a function that was specified at open time converts between native format and on-disk (serialized) format. This function is generated from the IDL file by the IDL compiler.

## Freeing Data

When fetching data, the Encoding Services (ES) allocate memory for the data structures that are returned. These services accept a structure, and use **rpc_sm_allocate()** to provide additional memory needed to hold the data.

The backing store library does not know what memory has been allocated, and therefore cannot free it. For fetch calls that are made from a server stub, this is not a problem, because the memory is freed automatically when the server call terminates. For fetch calls that are made from a nonserver, the programmer is responsible for freeing the memory.

Programs that call the fetch or store routines, such as **dce_db_fetch()**, outside of a server operation (for instance, if a server does some backing store initialization, or in a standalone program) must call **rpc_sm_enable_allocate()** first.

## Making or Retrieving Headers

The **dce_db_std_header_init()** routine initializes a standard backing store header from the values the caller provides in its arguments. It only places the values into the header, and does not write into the backing store file. The **dce_db_header_fetch()** routine retrieves the header of an object in the backing store.

## Performing Iteration

The following routines traverse all of the keys (name or UUID) in a backing store, iteratively. The order of retrieval of the keys is indeterminate; they are not sorted, nor are they necessarily returned in the order in which they were originally stored. It is strongly recommended to use the locking and unlocking routines, **dce_db_lock()** and **dce_db_unlock()**, whenever performing iteration.

| | |
|---|---|
| **dce_db_iter_start()** | This routine prepares for the start of iteration. |

| | |
|---|---|
| **dce_db_iter_next()** | This routine returns the key for the next item from a backing store that is indexed by name or by UUID. The **db_s_no_more** status code indicates that there are no more items. |
| **dce_db_iter_next_by_name()** | This routine returns the key for the next item only from a backing store that is indexed by name. Again, **db_s_no_more** indicates that no items remain. |
| **dce_db_iter_next_by_uuid()** | This routine returns the key for the next item only from a backing store that is indexed by UUID. Again, **db_s_no_more** indicates that no items remain. |
| **dce_db_iter_done()** | This routine is counterpart to **dce_db_iter_start()**, and should be called when iteration is done. |
| **dce_db_inq_count()** | This routine returns the number of items in a backing store. |

## Deleting Items from a Backing Store

The following routines delete an item from a backing store.

| | |
|---|---|
| **dce_db_delete()** | This routine deletes an item from a backing store that is indexed by name or by UUID. The key's type must match the flag that was used in **dce_db_open()**. |
| **dce_db_delete_by_name()** | This routine deletes an item only from a backing store that is indexed by name. |
| **dce_db_delete_by_uuid()** | This routine deletes an item only from a backing store that is indexed by UUID. |

To delete an entire backing store, ensure that the data file is not open, and remove it. There is only one file.

## Locking and Unlocking a Backing Store

The **dce_db_lock()** and **dce_db_unlock()** routines lock and unlock a backing store. If a backing store is already locked, **dce_db_lock()** provides an indication. A lock is associated with an open backing store's handle. The storage routines, **dce_db_store()**, **dce_db_store_by_name()**, and **dce_db_store_by_uuid()**, all acquire the lock before updating. Explicit use of locking is appropriate in some circumstances; for example, when reading or writing pairs (or multiples) of closely associated items in a backing store, or when using iteration.

The locks are advisory. It is possible to write a backing store even if it is locked, so if you want to rely upon the locks, you must always check them.

## Example of Backing Store Use

For a full example of backing store use, see the *z/OS DCE Application Development Guide: Introduction and Style*.

The following brief example shows a portion of a server that manages an office telephone directory. Following are the relevant structures defined in an IDL file:

```
typedef struct phone_record_s_t {
    [string,ptr] char       *name;
    [string,ptr] char       *email;
    [string,ptr] char       *phone;
    [string,ptr] char       *office;
} phone_record_t;

typedef struct phone_record_array_s_t {
                            unsigned32      count;
    [ptr,size_is(count)]    phone_record_t  *entry;
} phone_record_array_t;

typedef struct phone_data_s_t {
    dce_db_header_t h;
    phone_record_t  ph;
} phone_data_t;

/*
 * The following routine returns the entire contents of the
 * directory from the backing store by using the iteration routines.
 * First, the portion of the IDL file that defines the routine's
 * RPC format:
 */
[idempotent] void entire_phone_book(
    [in]        handle_t               h,
    [out]       phone_record_array_t  *e_array,
    [out]       error_status_t        *st
    );
```

Following is the routine itself, written in C:

```
/* global variables */
dce_db_handle__t  db_h;   /* handle to the phone book backing store
*/

/* Other routines are not shown here, including the routine that
 * opened the backing store.
 */

void
entire_phone_book(
    /* [in]  */ handle_t              h,    /* For RPC, but not used
                                             * here.  An ACL check
                                             * would use it.
                                             */
    /* [out] */ phone_record_array_t  *e_array,
    /* [out] */ error_status_t        *st
)
{
    uuid_t             *dbkey;
    phone_data_t        pd;
    unsigned32          i;
    error_status_t      st2;

    *st = error_status_ok;
    /* Lock before starting work, so that the backing store
     * does not change until after all the info has been returned.
     */
    dce_db_lock(db_h, st);
    /* Count the entries, so enough storage can be allocated */
    e_array->count = 0;
    dce_db_inq_count(db_h, &e_array->count, st);
    if (*st != error_status_ok) {
        dce_fprintf(stderr, *st); /* or some other treatment */
        dce_db_unlock(db_h, st);
        return;
    }
    if (e_array->count == 0) {    /* No items, nothing to do. */
```

```
        dce_db_unlock(db_h, st);
        return;
    }
    /* Allocate the space for the output. */
    e_array->entry = rpc_sm_allocate(
                     e_array->count*sizeof(e_array->entry[0]),st);
    if (*st != rpc_s_ok) {
        dce_fprintf(stderr, *st); /* or some other treatment */
        return
    }
    dce_db_iter_start(db_h, st);
    i = 0;
    while (TRUE) {
        /* Get the next key. */
        dce_db_iter_next(db_h, &dbkey, st);
        /* break when we've scanned the entire backing store */
        if (*st == db_s_no_more) break;
        /* Get the data associated with the next key. */
        dce_db_fetch_by_uuid(db_h, dbkey, (void *)&pd, st);
        if (*st != error_status_ok) {
            dce_fprintf(stderr, *st);
            /* Don't forget to stop iterating and unlock after an error. */
            dce_db_iter_done(db_h, &st2);
            dce_db_unlock(db_h, &st2);
            return;
        }
        /* Stick the item into the array to be returned when done. */
        e_array->entry[i].name   = strdup(pd.ph.name);
        e_array->entry[i].email  = strdup(pd.ph.email);
        e_array->entry[i].phone  = strdup(pd.ph.phone);
        e_array->entry[i].office = strdup(pd.ph.office);
        i++;
        /* The use of strdup() above is illustrative, but it is not
         * correct within a server, because the allocated memory is
         * never freed.  Correct code would involve the use of
         * rpc_sm_allocate().
         */
    }
    /* The iteration is finished. */
    dce_db_iter_done(db_h, st);
    dce_db_unlock(db_h, st);
}
```

# Part 2. Using the DCE Remote Procedure Call APIs

This part shows you how to use the Remote Procedure Call (RPC) APIs to create DCE RPC applications. You are introduced to the RPC model and RPC components and shown how to write an internationalized RPC application. Many other topics related to RPC application development are discussed, in addition to the details of the Interface Definition Language (IDL) and Attribute Configuration Language.

# Chapter 4. Developing a Simple RPC Application

This chapter first explains how to write an interface definition in the DCE RPC Interface Definition Language (IDL) and illustrates the basic features of IDL. As an example, we present an interface definition for **greet**, a very simple application that prints greetings from a client and a remote server. The remainder of the chapter describes how to develop, build, and run the **greet** client and server programs.

You can find the application source code for the **greet** example in the **usr/lpp/dce/examples/greet** directory.

The *z/OS DCE Application Development Guide: Introduction and Style* describes how to develop a DCE application using many of the features of DCE. The following chapters use the term RPC application to mean essentially the same thing, except in this context an RPC application concentrates on the features of the RPC technology, glossing over other DCE issues such as security, threads, and messaging. Since the RPC mechanism is the root technology for developing all DCE applications, the basic development approach is the same.

## The Remote Procedure Call Model

The Remote Procedure Call (RPC) model is a well-tested, industry-wide framework for distributing applications.

The RPC model is derived from the programming model of local procedure calls. The key difference is that an RPC executes a remote procedure, that is, one that is located in a *separate address space* from the calling code. The RPC model takes advantage of the fact that every procedure contains a procedure declaration which defines the interface (including call syntax and parameters) between the calling code and the called procedure. In an RPC this interface definition is the mechanism that makes called procedures remotely accessible.

An RPC application is divided into two parts: an RPC server, which offers one or more sets of remote procedures, and an RPC client, which makes remote procedure calls to RPC servers. The client and server generally (although not necessarily) reside on separate systems and communicate over a network.

The DCE RPC model is implemented via a runtime library of routines that perform the functions a distributed application needs, such as controlling network communications and finding servers for clients. This code is linked with client and server application code to form the RPC application.

Table 2 on page 40 shows the basic tasks of the client and server of an RPC application.

*Table 2. Basic Tasks of an RPC Application*

| Client Tasks | Server Tasks |
|---|---|
| | 1. Select network protocols |
| | 2. Register RPC interfaces |
| | 3. Advertise RPC interfaces and objects in the namespace |
| | 4. Listen for calls |
| 5. Find compatible servers that offer the procedures | |
| 6. **Call the remote procedure** | |
| 7. Establish a binding relationship with the server | |
| 8. Convert interface input arguments to network data representations | |
| 9. Transmit arguments to the server's runtime | |
| | 10. Receive call |
| | 11. Disassemble network data and convert arguments into local data |
| | 12. Locate and invoke the called procedure |
| | 13. **Execute the remote procedure** |
| | 14. Convert to network data representations the interface output arguments and possible return value |
| | 15. Transmit results to the client's runtime |
| 16. Receive results | |
| 17. Disassemble network data and convert arguments into local data | |
| 18. Pass to the calling code the results and return control to it | |

An RPC typically uses computing resources (such as processors, databases, devices, and services) dispersed on many systems. Some typical examples of RPC applications are :

- A calendar-management application that allows authorized users to access the personal calendars of other users
- A graphics application that processes data on central processors and displays the results on workstations
- A manufacturing application that shares changing information about assembly components among design, inventory, scheduling, and accounting programs located on different computers

## RPC Application Code

An RPC server or client contains application code, one or more RPC stubs, and a copy of the RPC runtime. RPC application code is the code written for a specific RPC application by the application developer. Application code implements and calls remote procedures, and also calls any RPC runtime routines the application needs. An RPC stub is an interface-specific code module that uses an RPC interface to pass and receive arguments. A server and a client contain complementary stubs for each RPC interface they share. The DCE RPC runtime manages communications for RPC applications. In addition, the DCE RPC runtime supports an Application Programming Interface (API) used by RPC application code to call runtime routines. These runtime routines enable RPC applications to set up their communications, manipulate information about servers, and perform optional tasks such as remotely managing servers and accessing security information.

Figure 4 on page 41 shows the relationship of application code, stubs, and the RPC runtime in the server and client portions of an RPC application.

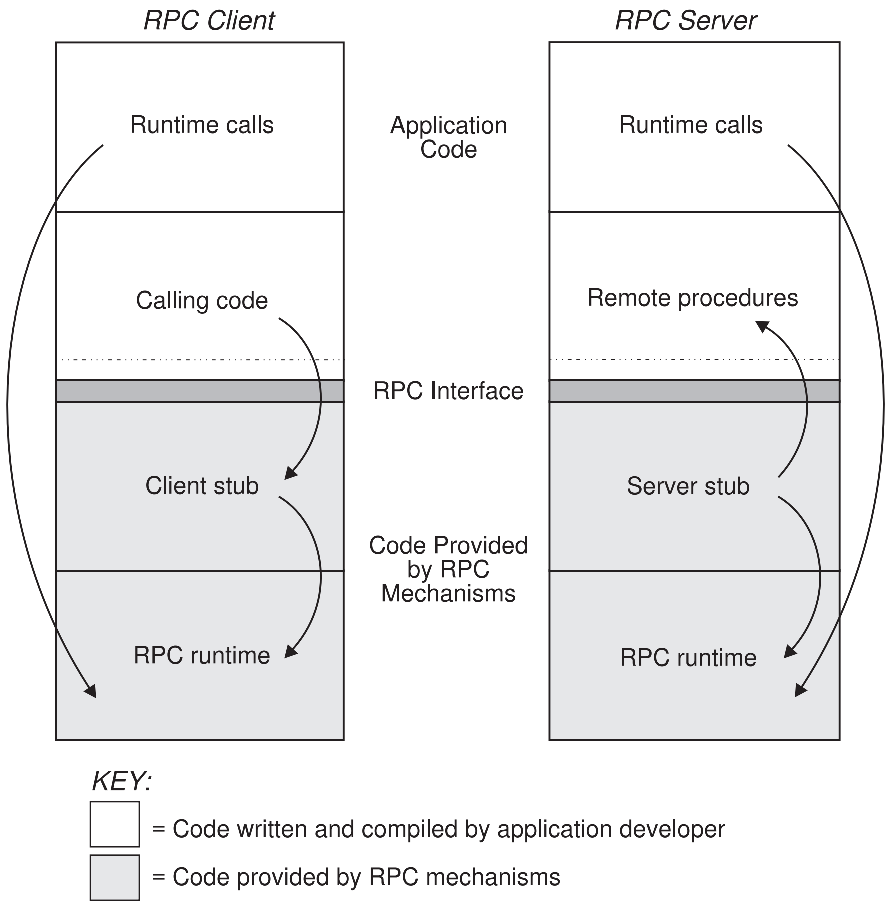


*Figure 4. The Parts of an RPC Application*

RPC application code differs for servers and clients. Minimally, server application code contains the remote procedures (also referred to as “manager code”) that implement one RPC interface, and the corresponding client contains calls to those remote procedures.

## Stubs

The stub performs basic support functions for remote procedure calls. For instance, stubs prepare input and output arguments for transmission between systems with different forms of data representation. The stubs use the RPC runtime to send and receive remote procedure calls. The client stub can also use the runtime to find servers for the client.

When a client application calls a remote procedure, the client stub first prepares the input arguments for transmission. The process for preparing arguments for transmission is known as “marshalling.” Marshalling converts call arguments into a byte-stream format and packages them for transmission. Upon receiving call arguments, a stub unmarshalls them. Unmarshalling is the process by which a stub disassembles incoming network data and converts it into application data using a format that the local system understands. Marshalling and unmarshalling both occur twice for each remote procedure call; that is, the client stub marshalls input arguments and unmarshalls output arguments, and the server stub unmarshalls input arguments and marshalls output arguments. Marshalling and unmarshalling permit client and server systems to use different data representations for equivalent data. For example, the client system can use ASCII characters and the server system can use EBCDIC characters as shown in Figure 5 on page 42.

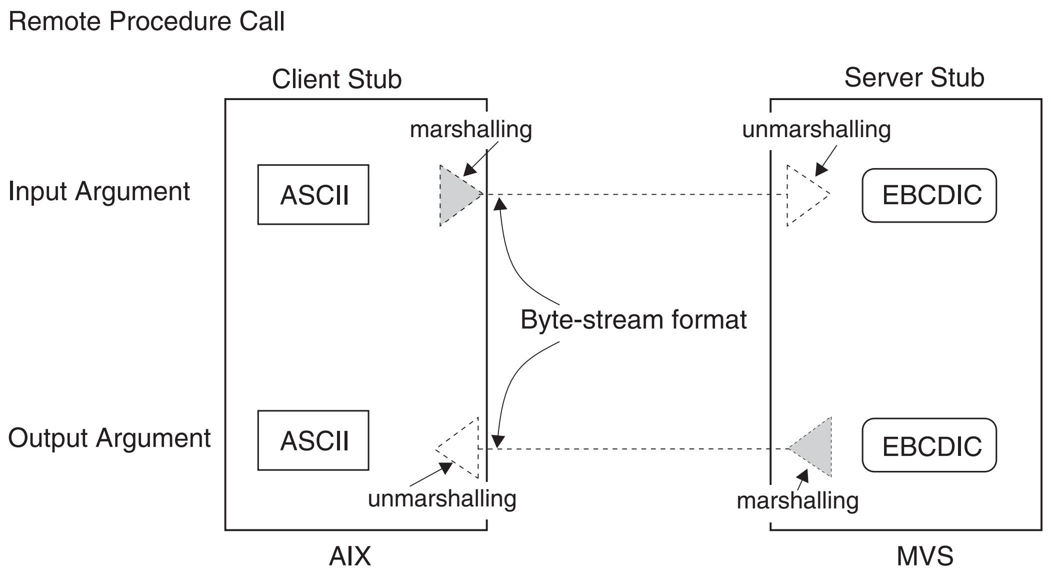


*Figure 5. Marshalling and Unmarshalling Between ASCII and EBCDIC Data*

The DCE IDL compiler (a tool for DCE application development) generates stubs by compiling an RPC interface definition written by application developers. The IDL compiler generates marshalling and unmarshalling routines for IDL data types specified in the interface definition (**.idl** file).

To build the client for an RPC application, a developer links client application code to the client stubs of all the RPC interfaces the application uses. To build the server, the developer links the server application code to the corresponding server stubs.

## The RPC Runtime

In addition to one or more RPC stubs, every RPC server and RPC client is linked with its own copy of the RPC runtime. Runtime operations perform tasks such as controlling communications between clients and servers and finding servers for clients on request. An interface's client and server stubs exchange arguments through their respective local RPC runtimes. The client runtime transmits remote procedure calls to the server. The server runtime receives the calls and dispatches each call to the appropriate server stub. The server runtime passes the call results to the client runtime. The DCE RPC runtime supports the RPC API used by RPC application code to call runtime routines.

Server application code must also contain server initialization code that calls RPC runtime routines when the server is starting up and shutting down. These routines can, for example, announce that the server is ready to receive requests or is ending request servicing. Client application code can also call RPC runtime routines, for example, to list all available servers with certain desired characteristics so the client can determine the best one to use. Server and client application code can also contain calls to RPC stub-support routines. Stub-support routines allow applications to perform programming tasks such as allocating and freeing memory storage. The marshalling and unmarshalling of data that is passed between the client and server is done with stub-support routines.

## RPC Application Components Working Together

Figure 6 shows the roles of application code, RPC stubs, and RPC runtimes during a remote procedure call.

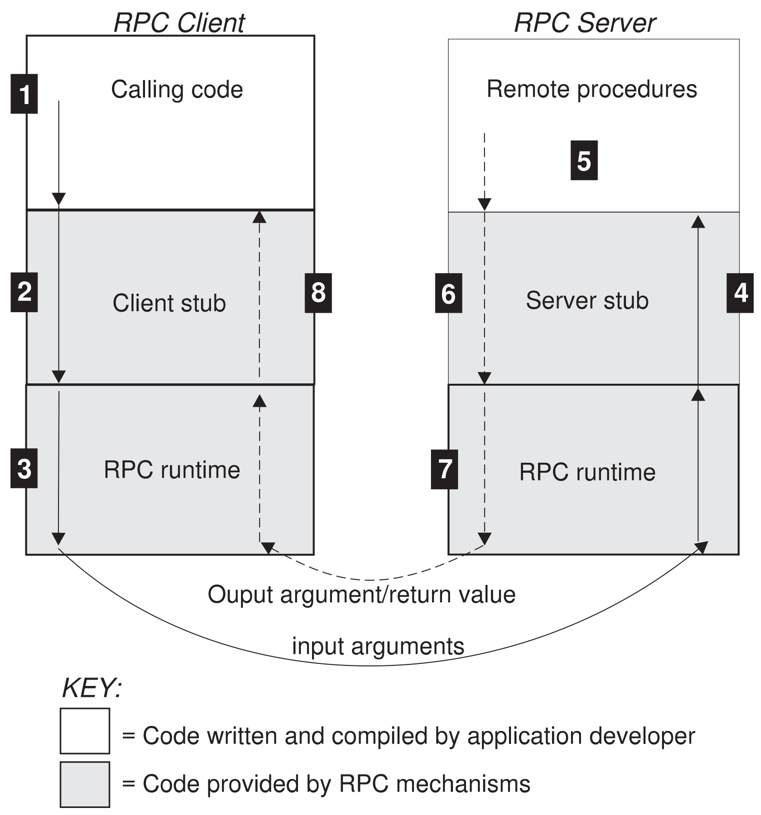


*Figure 6. Interrelationships During a Remote Procedure Call*

The following steps describe the interrelationships of the components of RPC applications, as shown in Figure 6:

1. The client's application code makes a remote procedure call, passing the input arguments to the stub for the particular RPC interface.
2. The client's stub marshalls the input arguments and dispatches the call to the client's RPC runtime.
3. The client's RPC runtime transmits the input arguments over the communications network to the server's RPC runtime, which dispatches the call to the server stub for the RPC interface of the called procedure.
4. The server's stub unmarshalls the input arguments and passes them to the called remote procedure.
5. The procedure executes and then returns any results (output arguments or a return value or both) to the server's stub.
6. The server's stub marshalls the results and passes them to the server's RPC runtime.
7. The server's RPC runtime transmits the results over the communications network to the client's RPC runtime, which dispatches them to the correct stub of the client.
8. The client's stub unmarshalls output arguments and passes them to the calling code.

Note that only steps one and five involve code that is written by the application developer; the rest of the steps are all handled by the RPC runtime.

## Overview of DCE RPC Development Tasks

The tasks involved in developing an RPC application resemble those involved in developing a local application. As an RPC developer, you perform the following basic tasks (these will be described in more detail later in this chapter):

1. Design your application as you normally would, but with the additional step of deciding which procedures you will be using will be remote procedures.
2. Generate a unique identifier (UUID) for each new planned interface by using the **uuidgen** command. (See 46 for the definition of a UUID.)
3. Write an interface definition file (named *interfacename***.idl** if using HFS) using the Interface Definition Language (IDL) that describes the RPC interfaces for the planned remote procedures. This step is similar to writing C language prototype **.h** files. (See “Writing an Interface Definition” on page 45.)
4. Use the DCE IDL compiler to generate the client and server stubs. The IDL Compiler invokes the C Compiler to create the stub object code. Figure 7 illustrates this task.

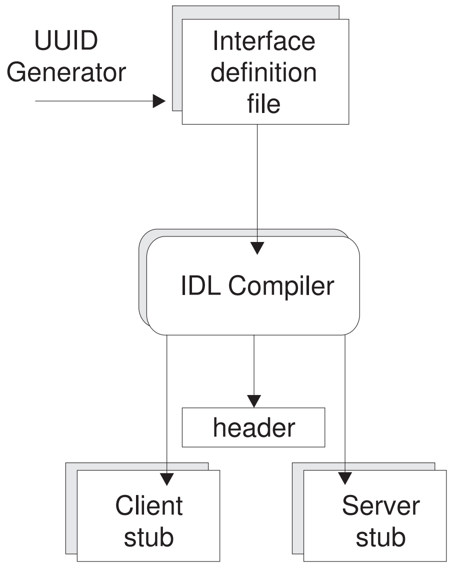


*Figure 7. Generating Stubs*

**Note:**

While the .idl and .acf files themselves are portable across all DCE platforms, the **.c, .h**, and **.o** stub files generated by the IDL compiler are not. Stub files used on the z/OS DCE platform **must** be generated with the z/OS DCE IDL compiler.

5. Write or modify application code using a compatible programming language; that is, a language that can be linked with C and can invoke C procedures, so the application code works with the stubs.

   Application code includes several kinds of code, as follows:

   a. Remote procedures (actual client and server application code functions)
   b. Remote procedure calls (for example, a call to check on which server is best to use for a procedure)

   c. Initialization code (calls to RPC stub-support or runtime routines)

   d. Any non-RPC code your application requires (for example, data display menus)

6. Generate object code from application code.

7. Create an executable client and server from the object files. Figure 8 illustrates this task.

   For the client, link object code of the client stub(s) and the client application with the RPC runtime and any other needed runtime libraries.

   For the server, link object code for the server stub(s), the initialization routines, and the set(s) of remote procedures with the RPC runtime and any other needed runtime libraries.

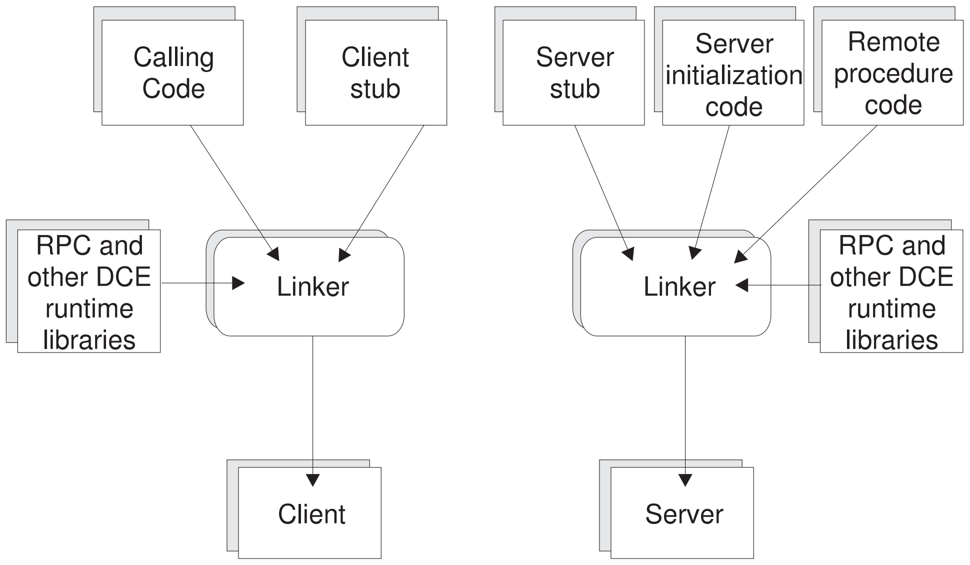

Calling Code
Client stub
Server stub
Server initialization code
Remote procedure code
RPC and other DCE runtime libraries
Linker
Linker
RPC and other DCE runtime libraries
Client
Server

*Figure 8. Building a Simple Client and Server*

8. After initial testing, distribute the new application by separately installing the server and client executable images on systems on the network.

## Writing an Interface Definition

Traditionally, calling code and called procedures share the same address space and are link-edited together. In an RPC application, the calling code and the called remote procedures are not linked; rather, they communicate indirectly through an RPC interface. An RPC interface is a logical grouping of operations, data types, and constants that serves as a unique network contract for a set of remote procedures. DCE RPC interfaces are compiled from formal interface definitions written by application developers using the DCE Interface Definition Language (IDL).

The first step in developing a distributed application is to write an interface definition file in IDL. The IDL compiler, **idl**, uses the information in an interface definition to generate a header file, a client stub file, and a server stub file. The IDL compiler produces header files in C and can produce stubs as C source files and/or as object files.

For some applications, you may also need to write an Attribute Configuration File (ACF) to accompany the interface definition. If an ACF exists, the IDL compiler interprets the ACF when it compiles the interface definition. Information in the ACF is used to modify the code that the compiler generates. (The **greet** example does not require an ACF.)

The remainder of this section briefly explains how to create an interface definition and gives simple examples of each kind of IDL declaration. For a detailed description of IDL, see Chapter 11, “Interface Definition Language” on page 221. For information on the IDL compiler, see the *z/OS DCE Command Reference*.

An IDL interface definition consists of a header and a body. The following example shows the interface definition for the **greet** application:

Each RPC interface contains in the header a Universal Unique Identifier (UUID), which is a hexadecimal number that can uniquely identify an entity. The UUID in the example above is `3d6ead56-06e3-11ca-8dd1-826901beabcd`. A UUID that identifies an RPC interface is known as an interface UUID. The **interface UUID** ensures that the interface can be uniquely identified across all possible network configurations. In addition to an interface UUID, each RPC interface contains major and minor version numbers. The example above is version `1.0`. Together, the interface UUID and version numbers form an interface identifier that identifies an instance of an RPC interface across systems and through time.

The interface body can contain the following constructs:

- Import declarations (not shown)
- Constant declarations (`REPLY_SIZE`)
- Type declarations (not shown)
- Operation declarations (`void greet(...);`)

IDL declarations resemble declarations in ANSI C. IDL is purely a declarative language, so, in some ways, an IDL interface definition is like a C header file. However, an IDL interface definition must specify the extra information that is needed by the remote procedure call mechanism. Most of this information is expressed via IDL attributes. IDL attributes can apply to types, to type members, to operations, to operation parameters, or to an interface as a whole. An attribute is represented in **[ ]** (brackets) before the item to which it applies. In the **greet.idl** example, the `[in, string]` attributes associated with the `client_greeting` array means the parameter is for input only and that the array of characters has the properties of strings.

A comment can be inserted at any place in an interface definition where white space is permitted. IDL comments, like C comments, begin with **/*** (a slash and an asterisk) and end with ***/** (an asterisk and a slash).

## RPC Interfaces Represent Services

The simplest RPC application uses only one RPC interface. However, an application can use multiple RPC interfaces, and frequently, an integral set of RPC interfaces work together as an RPC service. An RPC service is a logical grouping of one or more RPC interfaces. For example, you can write a calendar service that contains only a personal calendar interface or a calendar service that contains additional RPC interfaces such as a scheduling interface for meetings.

Different services can share one or more RPC interfaces. For example, an administrative support application can include an RPC interface from a calendar service.

An RPC interface exists independently of specific applications. That is, it can be implemented by by any client/server application that conforms to the interface definition, and all implementation of the same version of an interface will operate the same way. This makes it possible for clients from different implementations to call the same interface, and servers from different implementations to offer the same interface.

```
/*
 * greet.idl
 *
 * The "greet" interface.
 */

[uuid(3d6ead56-06e3-11ca-8dd1-826901beabcd),
version(1.0)]

interface greetif
{
    const long int REPLY_SIZE = 100;

    void greet(
        [in]            handle_t h,
        [in, string]    char client_greeting[],
        [out, string]   char server_reply[REPLY_SIZE]
    );
}
```

*Figure 9. GREET Interface Definition*

Figure 10 on page 47 shows the role of RPC interfaces in remote procedure calls. This client contains calling code that makes two remote procedure calls. The first is a remote procedure call to Procedure A. The second is a remote procedure call to Procedure B.

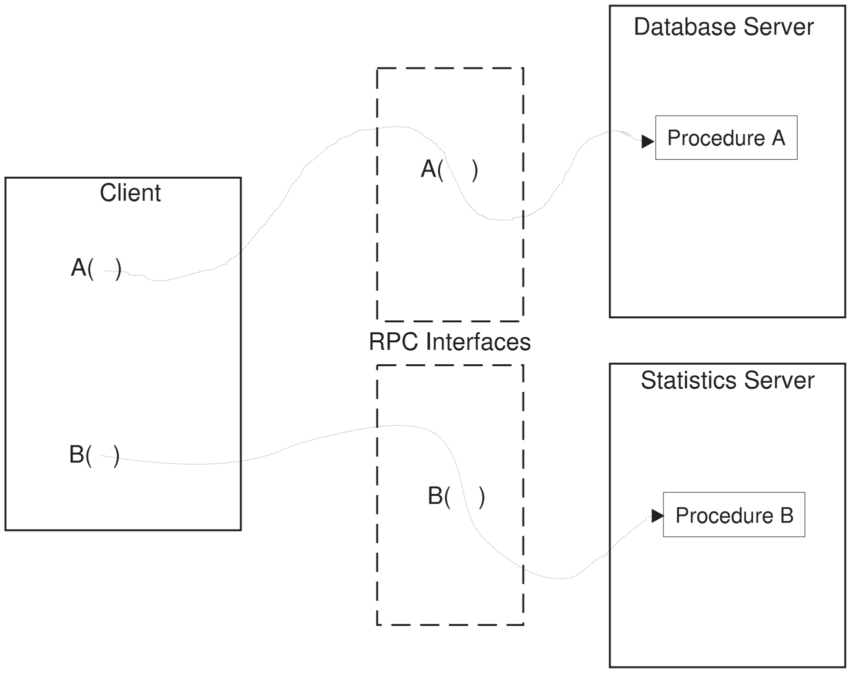


*Figure 10. Role of RPC Interfaces*

Any application may use interfaces A and B above if it conforms to the A and B's interface definitions (that is, incorporates A and B's interface specifications into its application code.)

Clients can use any practical combination of RPC interfaces, whether offered by the same or different servers. For this example, using a database access interface, a client on a graphics workstation can call a remote procedure on a database server to fetch data from a central database. Then, using a statistics interface, the client can call a procedure on another server on a parallel processor to analyze the data from the central database and return the results to the client for display.

## Generating an Interface UUID

The first step in building an RPC application is to generate a skeletal interface definition file and a UUID for the interface. Every interface in an RPC application must have a Universal Unique Identifier (UUID). When you define a new interface, you must generate a new UUID for it. (In an object-oriented application, every object and object type must also have a non-nil UUID.)

Typically, you run **uuidgen** with the **-i** option. The **-i** option produces a skeletal interface definition file and includes the generated UUID for the interface. For example:

```
 $ uuidgen -i > chess.idl
 $ cat chess.idl

[
uuid(443f4b20-a100-11c9-baed-08001e0218cb),
version(1)
]
interface INTERFACENAME {

}
```

The first part of the skeletal definition is the header, which specifies a UUID, a version number, and a name for the interface. The last part of the definition is **{ }** the body, (an empty pair of braces); import, constant, type, and operation declarations go between these braces.

By convention, the names of interface definition files end with the suffix **.idl**. The IDL compiler constructs names for its output files based on the name of the interface definition file and uses the following default suffixes:

- **.h** for header files
- **_cstub.o** for client stub files
- **_sstub.o** for server stub files

For example, compilation of a **chess.idl** interface definition file would produce a **chess.h** header file, a **chess_cstub.o** client stub file, and a **chess_sstub.o** server stub file. (The IDL compiler creates C language intermediate files and by default invokes the C compiler to produce object files, but it can also retain the C language files.)

For more information on the UUID generator, see the *z/OS DCE Command Reference*.

## Naming the Interface

After you have used **uuidgen** to generate a skeletal interface definition, replace the dummy string **INTERFACENAME** with the name of your interface.

By convention, the name of an interface definition file is the same as the name of the interface it defines, with the suffix **.idl** appended. For example, the definition for a **bank** interface would reside in a **bank.idl** interface definition file, and if the application required an ACF, its name would be **bank.acf**.

The IDL compiler incorporates the interface name in identifiers it constructs for various data structures and data types, so the length of an interface name can be at most 17 characters. (Most IDL identifiers have a maximum length of 31 characters.)

## Specifying Interface Attributes

Interface attributes are defined within **[ ]** (brackets) in the header of the interface definition. The definition for any remote interface needs to specify the **uuid** and **version** interface attributes, so these are included in the skeletal definition that **uuidgen** produces.

If an interface is exported by servers on well-known endpoints, these endpoints must be specified via the **endpoint** attribute. Interfaces that use dynamic endpoints do not have this attribute. (A well-known endpoint is a stable address on the host, while a dynamic endpoint is an address that the RPC runtime requests for the server.)

The interface definition language can be used to specify procedure prototypes for any application, even if the procedures are never used remotely. If all of the procedures of an interface are called only locally and never remotely, the interface can be given the **local** attribute. Since local calls do not have any network overhead, the **local** attribute causes the compiler to generate only a header file, not stubs, for the interface.

## Import Declarations

The IDL **import** declaration specifies another interface definition whose types and constants are used by the importing interface.

The **import** declaration allows you to collect declarations for types and constants that are used by several interfaces into one common file. For example, if you are defining two database interfaces named **dblookup** and **dbupdate**, and these interfaces have many constants in common, you can declare those constants in a **dbconstants.idl** file and import this file in the **dblookup.idl** and **dbupdate.idl** interface definitions. For example:

```
import "dbconstants.idl";
```

By default, the IDL compiler resolves relative path names of imported files by looking first in the current working directory and then in the system IDL directory. The **-I** option of the IDL compiler allows you to specify additional directories to search. You can thereby avoid putting absolute path names in your interface definitions. For example, if an imported file has the absolute path name **/dbproject/src/dbconstants.idl**, then the IDL compiler option **-I/dbproject/src** allows you to import the file by its leaf name, **dbconstants.idl**.

## Constant Declarations

The IDL **const** declaration allows you to declare integer, Boolean, character, string, and null pointer constants, some of which are shown in the following examples:

```
const short TEN = 10;
const boolean VRAI = TRUE;
const char* JSB = "Johann Sebastian Bach";
```

## Type Declarations

To support application development in a variety of languages and to support the special needs of distributed applications, IDL provides an extensive set of data types, including the following:

- Simple types, such as integers, floating-pointing numbers, characters, Booleans, and the primitive binding-handle type **handle_t** (equivalent to **rpc_binding_handle_t**)
- Constructed types, such as strings, structures, unions, arrays, pointers, and pipes
- Predefined types, including ISO international character types and the error status type **error_status_t**

The IDL **typedef** declaration lets you give a name to any types you construct.

The general form of a type declaration is

**typedef [***type_attribute***,...]** *type_specifier type_declarator***,...;**

where the bracketed list of type attributes is optional. The *type_specifier* specifies a simple type, a constructed type, a predefined type, or a type previously named in the interface. Each *type_declarator* is a name for the type being defined. As in C, arrays and pointers are declared by the *type_declarator* constructs **[ ]** (brackets) and an ***** (asterisk) rather than by *type_specifier* constructs.

The following type declaration defines **integer32** as a name for a 32-bit integer type:

```
typedef long integer32;
```

The *type_specifier* constructs for structures and unions permit the application of attributes to members. In the following example, one member of a structure is a conformant array (an array without a fixed upper bound), and the **size_is** attribute names another member of the structure that at runtime provides information about the size of the array:

```
typedef struct {
     long dsize;
     [size_is(dsize)] float darray[];
     } dataset;
```

## Operation Declarations

Operation declarations specify the signature of each operation in the interface, including the operation name, the type of data returned (if any), and the types of all parameters passed (if any) in a call.

The general form of an operation declaration is

**[***operation_attribute***,...]** *type_specifier operation_identifier* **(***parameter_declaration* **,...);**

where the bracketed list of operation attributes is optional. Among the possible attributes of an operation are **idempotent**, **broadcast**, and **maybe**, which specify semantics to be applied by the RPC runtime mechanism when the operation is called. If an operation when called once can safely be executed more than once, the IDL declaration of the operation may specify the **idempotent** attribute; **idempotent** semantics allow remote procedure calls to execute more efficiently by letting the underlying RPC mechanism retry the procedure if it deems it necessary. **Broadcast** semantics specify that the operations is always to be broadcast to all hosts on a network. **Maybe** specifies that the caller of the operation does not require and will not receive any response.

The *type_specifier* specifies the type of data returned by the operation.

The *operation_identifier* names the operation. Although operation names are arbitrary, a common convention is to use the name of an interface as a prefix for the names of its operations. For example, a **bank** interface may have operations named **bank_open**, **bank_close**, **bank_deposit**, **bank_withdraw**, and **bank_balance**.

Each *parameter_declaration* in an operation declaration declares a parameter of the operation. A *parameter_declaration* takes the following form:

**[***parameter_attribute***, ...]** *type_specifier parameter_declarator*

Every parameter attribute must have at least one of the parameter attributes **in** or **out** to specify whether the parameter is passed as an input, as an output, or in both directions. The *type_specifier* and *parameter_declarator* specify the type and name of the parameter.

Output parameters must be passed by reference and therefore must be declared as pointers via the pointer operator ***** (an asterisk) or an array.

If you want an interface to always use explicit binding handles, the first parameter of each operation declaration must be a binding handle, as in the following example:

```
void greet(
    [in]            handle_t h,
    [in, string]    char client_greeting[],
    [out, string]   char server_reply[REPLY_SIZE]
);
```

However, if you want applications to use an implicit handle or use automatic binding (ACF features) for some or all procedures, operation declarations must not have binding handle parameters in the interface definition:

```
void greet_no_handle(
    [in, string]    char client_greeting[],
    [out, string]   char server_reply[REPLY_SIZE]
);
```

This form of operation declaration is the most flexible because applications can always specify explicit, implicit, or automatic binding handles through an ACF.

## Running the IDL Compiler

After you have written an interface definition, run the IDL compiler to generate header and stub files. The compiler offers many options that, for example, allow you to choose what C compiler or C preprocessor commands are run, what directories are searched for imported files, which of the possible output files are generated, and how the output files are named.

When you compile the definition of a remote interface, you must ensure that the system IDL directory is among those that the compiler searches for imported files because any remote interface implicitly imports **rpc.idl**.

The **greet.idl** interface definition (shown in Figure 9 on page 47) can be compiled by the following command:

```
% idl greet.idl
```

This compilation produces a header file, **greet.h**; a client stub file, **greet_cstub.o**; and a server stub file, **greet_sstub.o**. For complete information on running the IDL compiler, see the *z/OS DCE Command Reference*.

## Writing the Client Code

This section describes the client program for the **greet** application, whose interface definition was shown earlier in this chapter.

The client performs the following major steps:

1. It calls **rpc_string_binding_compose()** to create a string binding from the components of binding information specified (protocol sequence, host network address, etc.) These components are entered by the user at runtime.
2. It calls **rpc_binding_from_string_binding()** to convert the string binding into a server binding handle.

3. It calls the **greet_rpc** remote procedure with a string greeting.
4. It prints the reply from the server.

The **greet_client.c** module is as follows:

```
/*
 * greet_client.c
 *
 * Client of "greet" interface.
 */

#include <stdio.h>
#include <locale.h>
#include <dce/dce_error.h>
#include "greet.h"

int main(int argc, char *argv[])
{
    handle_t                 h;
    unsigned long            st;
    int                      error_inq_st;
    dce_error_string_t       error_text;
    idl_char                 *string_binding, *protseq, *hostid, *endpoint;
    static idl_char          nil_string[] = "";
    int                      i, MAX_PASS;
    char                     reply[STR_SZ];

    setlocale(LC_ALL, "");

    if (argc != 5)  {
        fprintf(stderr, "Usage: %s protseq hostid endpoint passes\n", argv[0]);
        fflush(stderr);
        exit (1);
    }

    protseq  = (idl_char *) argv[1];
    hostid   = (idl_char *) argv[2];
    endpoint = (idl_char *) argv[3];

    rpc_string_binding_compose(nil_string,
                 protseq, hostid, endpoint,
                 nil_string, &string_binding, &st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr,"Can't compose string binding - %s\n", error_text);
        fflush(stderr);
        exit(1);
    }

    rpc_binding_from_string_binding(string_binding, &h, &st);
    if (st != error_status_ok)  {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr, "Cannot get binding from string binding %s - %s\n",
                string_binding,error_text);
        fflush(stderr);
        exit(1);
    }

    printf("Bound to %s\n",string_binding);
    fflush (stdout);

    MAX_PASS= atoi(argv [4]);

         for (i=1; i <= MAX_PASS; i++) {
             greet_rpc(h, "Hello Server !", reply);
```

```
            printf("The Greet Server said: %s\n", reply);
            fflush(stdout);
        }

return(0);
}
```

The module first includes **greet.h**, the header file for the **greet** interface generated by the IDL compiler.

The client program checks the status after each call to the RPC runtime. If it is not **error_status_ok**, **dce_error_inq_text()** is called and the error message is printed.

As specified in the **greet.idl** interface definition, the **greet** application uses implicit binding handles. Before making the remote call to the server, the client initializes a server binding handle by having the runtime convert string binding information to a binding handle. The client then assigns this binding handle to a global variable, and uses it to make the remote call.

## Writing the Server Code

The following subsections describe the server program for the **greet** application.

The **greet** server program has two user-written modules:

- The **greet_server.c** module contains the server **main** function and performs the initialization and registration required to export the **greet** interface.
- The **greet_manager.c** module contains the code that actually implements the **greet** operation.

## The greet_server.c Source Code

Most applications should use the DCE convenience routines for server initialization routines (routines that begin with **dce_server**) to prepare servers to listen for remote procedure calls. These routines are simple to use, prepare a server so that **dced** can manage it, and they allow enough flexibility to do most typical initializations. However, for detailed control, applications can also use the lower-level RPC API to do server initialization. In this chapter we describe how to use the RPC API for server initialization.

In this section, the **greet_server.c** module is described and shown in successive pieces.

**Including idl-Generated Headers and Selecting Protocol Sequences:** Like **greet_client.c**, the **greet_server.c** module includes **greet.h** so that constants, data types, and procedure prototypes are available in the application.

The server then calls **rpc_server_use_all_protseqs()** to obtain endpoints on which to listen for remote procedure calls. This call also specifies the maximum number of calls this server can accept at the same time (in this case, 5) with the MAX_CONCURRENT_CALLS parameter.

```
/*
 * greet_server.c
 *
 * Main program (initialization) for "greet" server.
 */
#pragma runopts(stack(12K, 4K, ANY,KEEP))

#include <stdio.h>
#include <locale.h>
#include <dce/dce_error.h>
```

```
#include <dce/exc_handling.h>
#include "greet.h"

#define MAX_CONCURRENT_CALLS 5

/* In the first part of the main function, the server calls
   rpc_server_use_all_protseqs() to use all protocol
   sequences that are supported on its host both by the
   runtime library and the operating system.             */

int main (int argc, char *argv⅛‘)
{
    rpc_binding_vector_p_t  bvec;
    unsigned long           st;
    int                     error_inq_st;
    dce_error_string_t      error_text;
    idl_boolean             validfamily;
    idl_char                *string_binding;
    int                     i;

    setlocale(LC_ALL, "");

    if (argc != 1) {
        fprintf(stderr, "Usage: %s\n", argv⅛0‘);
        fflush(stderr);
        exit(1);
    }

    /* Calling rpc_server_use_all_protseqs to obtain an endpoint
       for each protocol sequence supported by the RPC runtime and
       the operating system */

    rpc_server_use_all_protseqs(MAX_CONCURRENT_CALLS, &st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr, "Cannot register protocol seqs - %s\n", error_text);
        fflush(stderr);
        exit(1);
    }
```

**Registering the Interface:** The server calls **rpc_server_register_if()**, supplying its interface specifier (defined in **greet.h**), to register each interface with the RPC runtime:

```
    /*
     * Register interface with RPC runtime.
     */
    rpc_server_register_if(greetif_v1_0_s_ifspec, NULL, NULL,
       &st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr,"Cannot register interface - %s\n",error_text);
        fflush(stderr);
        exit(1);
    }
```

**Obtaining the Server's Binding Handles:** To obtain a vector of binding handles that it can use when registering endpoints, the server calls **rpc_server_inq_bindings()**: It then converts a binding handle to a string binding using **rpc_binding_to_string_binding()**. It prints the string binding and then frees it after use.

```
    /* Calling rpc_server_inq_bindings to obtain a vector of
       binding handles that can be used to register the server's
       endpoint. The server then obtains, prints, and frees a
       string binding */

    rpc_server_inq_bindings(&bvec, &st);
    if (st != error_status_ok)  {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr,"Cannot inquire bindings - %s\n",error_text);
        fflush(stderr);
        exit(1);
    }

    printf("Bindings:\n");
    fflush(stdout);

    for (i = 0; i < bvec->count; i++)  {
         rpc_binding_to_string_binding(bvec->binding_h[i],
              &string_binding, &st);
         printf("%s\n", (char *)string_binding);
         fflush(stdout);
         rpc_string_free(&string_binding, &st);
    }
```

**Listening for Calls:** To begin listening for remote procedure call requests, the server calls **rpc_server_listen()**.

```
    /*  To begin listening for RPC requests, the server calls
        rpc_server_listen.  This call is placed within the TRY of a
        TRY, CATCH_ALL, ENDTRY sequence, so that if the server receives
        a signal while it is listening, the CATCH_ALL code will allow
        the server to shut down gracefully. */

    TRY {
         printf("Listening...\n");
         fflush(stdout);
         rpc_server_listen(MAX_CONCURRENT_CALLS, &st);
         if (st != error_status_ok) {
             dce_error_inq_text(st, error_text, &error_inq_st);
             fprintf(stderr, "Error: %s\n",error_text);
             fflush(stderr);
         }
    }

    CATCH_ALL {
         printf("Server GREET shutting down\n");
         fflush(stdout);
    }

    ENDTRY;

return(0);
}
```

## The greet_manager.c Source Code

The **greet_manager.c** module includes **greet.h** and it also defines the routine **greet_rpc**, as follows:

```
/*
 * greet_manager.c
 *
 * Implementation of "greet" interface.
 */

#include <stdio.h>
#include "greet.h"

void greet_rpc(handle_t h,
               char *client_greeting,
               char *server_reply)
{
       printf("The client says: %s\n", client_greeting);
       fflush(stdout);
       strncpy(server_reply, "Hi client !", STR_SZ);
}
```

## Building the greet Programs

The client side of the **greet** application is the **greet_client** program, which is built from the following:

- The user-written **greet_client.c** client module
- The IDL-compiler-generated **greet_cstub.o** client stub module
- Libraries for the RPC runtime

The server side of the **greet** application is the **greet_server** program, which is built from the following:

- The user-written **greet_server.c** server module
- The user-written **greet_manager.c** manager module
- The IDL-compiler-generated **greet_sstub.o** server stub module
- Libraries for the RPC runtime

These programs can be built by **make** with a makefile such as the following:

```
PROGRAMS = greet_server greet_client
INCLUDES = greet.h
IDLFILES = greet.idl
ILIST    = Makefile README greet_server.c greet_manager.c \
           greet_client.c greet.idl
greet_server_OFILES = greet_sstub.o greet_server.o greet_manager.o
greet_client_OFILES = greet_cstub.o greet_client.o
IDLFLAGS = -no_cpp -keep c_source
IDIR     = /examples/greet/
LIBS = -ldce
greet_server_OFILES : greet.h
greet_client_OFILES : greet.h
.include <${RULES_MK}>
```

## Running the greet Programs

Running the **greet** application involves starting the server program and then running the client program. For more information, see the *z/OS DCE Application Development Guide: Introduction and Style*. The following message is printed on the server's host:

```
The client says: Hello, Server!
```

The following reply is printed on the client's host:

```
The Greet Server said: Hi, client!
```

The server program can be terminated at any time by a quit signal, which on many systems can be generated by <**Ctrl-c**>:

<**Ctrl-c**>

$

When applications such as **greet** execute, many errors can occur that have nothing to do with your own code. In general, errors that occur when a remote procedure call executes are reported as exceptions. For example, exceptions that the client side of **greet_client** could raise if the server suddenly and unexpectedly halts include (but are not limited to) **rpc_x_comm_failure** and **rpc_x_call_timeout**. Other ways to respond to these errors are available through the **comm_status** and **fault_status** attributes in an interface definition or attribute configuration file. Explanations of these attributes appear in Chapter 12, “Attribute Configuration Language” on page 285. Also, see Chapter 10, “Topics in RPC Application Development” on page 173, which explains the guidelines for error handling.

Part 3, “Using the DCE Threads APIs” on page 313 contains information about the macros (such as those specified by **TRY**, **CATCH**, and **ENDTRY** statements) for exception handling. If an exception occurs that the client application does not handle, it causes the client to terminate with an error message. The client's termination could include a core dump or other system-dependent error reporting method. Detailed explanations of RPC status codes and RPC exceptions are in the *z/OS DCE Messages and Codes* book.

# Chapter 5. RPC Fundamentals

The DCE RPC call environment essentially behaves like a local call environment. However, the remoteness (that is, distribution among different address spaces) of the calling code to the called procedure does have some special implications. Therefore, a remote procedure call may not always behave exactly like a local procedure call.

**Note:**

RPC function in z/OS DCE differs slightly from that available in OSF DCE. The following RPC routines are IBM added APIs and may not be available in other DCE environments:

- rpc_err_get_uuid()
- rpc_err_reset_uuid()
- rpc_err_test_uuid()
- rpc_err_user_exit()
- rpc_err_ss_destroy_callee_context().

## Direct Implications of Remoteness

Distributed applications have the following implications:

- Server/client relationship: ***binding***

  Like a local procedure call, a remote procedure call depends on a static relationship between the calling code and the called procedure. In a local application, this relationship is established by linking the calling and called code. Linking gives the calling code access to the address of each procedure to be called. Enabling a remote procedure call to go to the right procedure requires a similar relationship (called a *binding*) between a client and a server. A ***binding*** is a temporary relationship that depends on a communications link between the client and server RPC runtimes. A client establishes a binding over a specific protocol sequence to a specific host system and endpoint.

- Lack of shared memory

  The calling code and called remote procedure reside in different address spaces, generally on separate systems. The calling and called code cannot share global variables or other global program states such as open files. All data shared between the caller and the called remote procedure must be specified as procedure parameters. Unlike a local procedure call that commonly uses the call-by-reference passing mechanism for input/output parameters, remote procedure calls with input/output parameters have copy-in/copy-out semantics due to the differing address spaces of the calling and called code. These two passing mechanisms are only slightly different, and most procedure calls are not sensitive to the differences between call-by-reference and copy-in/copy-out semantics.

- Independent Failure

  Distributing a calling program and the called procedures to physically separate machines increases the complexity of procedure calls. Remoteness introduces issues such as a remote system failure, communications links, naming and binding issues, security problems, protocol incompatibilities, performance and diagnosis. Such issues can require error handling that is unnecessary for local procedure calls. Also, as with local procedure calls, remote procedure calls are subject to run time errors that arise from the procedure call itself.

This chapter discusses the following topics:

- Universal unique identifiers

- Communications protocols
- Binding information
- Endpoints
- Run time semantics
- Communications failures
- Scalability
- RPC objects

## Universal Unique Identifiers

A Universal Unique Identifier (UUID) is a hexadecimal number. Each UUID contains information, including a timestamp and a host identifier.

Applications use UUIDs to identify many kinds of entities. DCE RPC identifies several kinds of UUIDs according to the kind of entities each identifies:

- ***Interface UUID***

  An interface UUID identifies a specific RPC interface. It is declared in an RPC interface definition and is a required element of the interface. For example:

  ```
  uuid(2fac8900-31f8-11ca-b331-08002b13d56d),
  ```

- ***Object UUID***

  An object UUID is a UUID that identifies an entity for an application; for example, a resource, a service, or a particular instance of a server. An application defines an RPC object by associating the object with its own UUID known as an ***object UUID***. The object UUID exists independently of the object, unlike an interface UUID. If different servers offer the same RPC object, the servers typically use different object UUIDs to identify it. A server usually generates UUIDs for its objects as part of initialization. A given object UUID is meaningful only while a server is offering the corresponding RPC object to clients.

  **Note:** To distinguish a specific use of an object UUID, a UUID is sometimes labeled for the entity it identifies. For example, an object UUID that is used to identify a particular instance of a server is known as an ***instance UUID***.

- ***Type UUID***

  A ***type UUID*** identifies a class of RPC objects and an associated manager. A manager is the set of remote procedures that puts into effect the RPC interface for objects of that type.

Servers can create object and type UUIDs by calling the **uuid_create()** routine.

## Communications Protocols

A communications link depends on a set of communications protocols. A communications protocol is a clearly defined set of operational rules and procedures for communications.

Communications protocols include a ***transport protocol*** (from the Transport Layer of the OSI network architecture) such as the ***Transmission Control Protocol*** (***TCP***) or the ***User Datagram Protocol*** (***UDP***); and the corresponding ***network protocol*** (from the OSI Network Layer), such as the Internet Protocol (IP).

For an RPC client and server to communicate, their RPC runtimes must use at least one identical communications protocol, including a common RPC protocol, transport protocol, and network protocol. An ***RPC protocol*** is a communications protocol that supports the semantics of the DCE RPC API and runs over specific combinations of transport and network protocols. DCE RPC provides two RPC protocols: the connectionless RPC protocol and the connection-oriented RPC protocol.

- ***Connectionless (Datagram) RPC protocol***

  This protocol runs over a connectionless transport protocol such as UDP. The connectionless protocol supports broadcast calls.

- ***Connection-oriented RPC protocol***

  This protocol runs over a connection-oriented transport protocol such as TCP.

Each binding uses a single RPC protocol and a single pair of transport and network protocols. Only certain combinations of communications protocols are functionally valid (are actually useful for interoperation); for instance, the RPC connectionless protocol cannot run over connection-oriented transport protocols such as TCP. DCE RPC supports the following combinations of communications protocols:

- RPC connection-oriented protocol over the Internet Protocol Suite, Transmission Control Protocol (TCP/IP)
- RPC connectionless over the Internet Protocol Suite, User Datagram Protocol (UDP/IP)

## Binding Information

In general terms, ***binding information*** is information about one or more potential binding instances. Binding information includes a set of information that identifies a server to a client or a client to a server. Each instance of binding information contains all or part of a single address. The RPC runtime maintains binding information for RPC servers and clients. To make a specific instance of locally maintained binding information available to a given server or client, the runtime creates a local reference known as a ***binding handle***. Servers and clients use binding handles to refer to binding information in runtime calls or remote procedure calls. A server obtains a complete list of its binding handles from its RPC runtime. A client obtains one binding handle after each binding to a server from its RPC runtime. Figure 11 illustrates a binding.

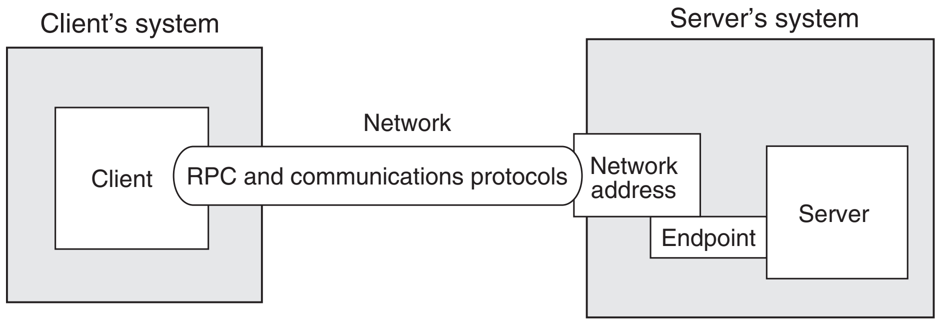


*Figure 11. A Binding*

Binding information includes the following components:

- **A protocol sequence:** A valid combination of communications protocols represented by a character string. Each protocol sequence typically includes a network protocol, a transport protocol, and an RPC protocol that works with them. For example, RPC connection-oriented protocol over TCP/IP.

An RPC server tells the runtime which protocol sequences to accept when listening for calls to the server via its binding information that contains those protocol sequences.

- **Network addressing information:** the network address and the endpoint of a server.
  - The network address identifies a specific host system on a network. The format of the address depends on the network protocol portion of the protocol sequence.
  - The *endpoint* acts as the address of a specific server instance on the host system. The format of the endpoint depends on the transport protocol portion of the protocol sequence. For each protocol sequence a server instance uses, it requires a unique endpoint. A given endpoint can be used by only one server per system, on a first-come, first-served basis.
- **Transfer Syntax:** a set of encoding rules used for transmitting data over a network and for converting application data to and from different local data representations. The server's RPC runtime must use a ***transfer syntax*** that matches one used by the client's RPC runtime. This shared transfer syntax enables communications between systems that represent local data differently. DCE RPC uses a single transfer syntax, Network Data Representation (NDR). NDR encodes data into a byte stream for transmission over a network. A transfer syntax, such as NDR, enables machines with different formats to exchange data successfully. (The DCE RPC communications protocols support the negotiation of transfer syntax. However, at present, the outcome of a transfer-syntax negotiation is always NDR.)
- RPC protocol version numbers: The client and server runtimes must use compatible versions of the RPC protocol specified by the client in the protocol sequence. The major version number of the RPC protocol used by the server must equal the specified major version number. The minor version number of the RPC protocol used by the server must be greater than or equal to the specified minor version number. This means that a client at version 1.0, level 0 can communicate with a server at version 1.1, but not with a server at version 2.0.

## Server Binding Information

Binding information for a server is known as ***server binding information***. A binding handle that refers to server binding information is known as a server binding handle. The use of server binding handles differs on servers and clients.

- **On a server**

  Servers use a list of server binding handles. Each represents one way to establish a binding with the server. Before exporting binding information to a namespace, a server tells the RPC runtime which RPC protocol sequences to use for the RPC interfaces the server supports. For each protocol sequence, the server runtime creates one or more server binding handles. Each server binding handle refers to binding information for a single potential binding, including a protocol sequence, a network (host) address, an endpoint (server address), a transfer syntax, and an RPC protocol version number.

- **On a client**

  A client uses a single server binding handle that refers to the server binding information the client needs for making one or more remote procedure calls to a given server. Server binding information on a client contains binding information for one potential binding.

  On a client, server binding information always includes a protocol sequence and the network address of the server's host system. However, sometimes a client obtains binding information that lacks an endpoint, resulting in a partially bound binding handle. A ***partially bound binding handle*** corresponds to a system, but not to a particular server instance. When a client makes a remote procedure call using a partially bound binding handle, the client runtime gets an endpoint either from the interface specification or from the endpoint map on the server's system. Adding the endpoint to the server binding information results in a ***fully bound binding handle***, which contains an endpoint and corresponds to a specific server instance.

## Defining a Compatible Server

*Compatible binding information* identifies a server whose communications capabilities (RPC protocol and protocol major version number, network and transport protocols, and transfer syntax) are compatible with those of the client. Compatible binding information is sufficient for establishing a binding. However, binding information is insufficient for ensuring that the binding is to a compatible server; that is, a server that also offers the requested RPC interface and RPC object (if any).

A compatible server is a server that meets the following criteria:

- Offers the requested RPC interface
- Offers the requested RPC object (if any)
- Shares a common communications environment with the client; that is, the client and server RPC runtimes must support the following:
  - At least one common pair of transport and network protocols, such as UDP/IP or TCP/IP
  - At least one common transfer syntax, such as NDR
  - The same DCE RPC protocol (connection-oriented or connectionless protocol) and RPC protocol major version number

The additional information that a client imposes on the RPC runtime includes an RPC interface identifier and an object UUID, as follows:

- Interface identifier: The interface UUID and version numbers of an RPC interface:
  - Interface UUID: The interface UUID, unlike the interface name, clearly identifies the RPC interface across time and space.
  - Interface version number: The combined major and minor version numbers identify one generation of an interface.

    Version numbers allow multiple versions of an RPC interface to coexist. Strict rules govern valid changes to an interface and determine whether different versions of an interface are compatible. For a description of these rules see Chapter 11, “Interface Definition Language” on page 221.

    The runtime uses the version number of an RPC interface to decide whether the version offered by a given server is compatible with the version requested by a client. The offered and requested interface are compatible under the following conditions:

    - The interface requested by the client and the interface offered by the server have the same major version number.
    - The interface requested by the client has a minor version number less than or equal to that of the interface offered by the server.
- An object UUID: A Universal Unique Identifier that identifies a particular object.

  An ***object*** is a distinct computing resource, such as a particular database, a specific RPC service that a remote procedure can access, and so on; for example, personal calendars are RPC objects to a calendar service. Accessing an object requires including its object UUID with the binding information used for establishing a binding. A client can request a specific RPC object when requesting new binding information, or the client can ask the runtime to associate an object UUID with binding information the client already has available.

  Sometimes the object UUID is the *nil UUID*, which contains only zeros, **00000000-0000-0000-0000-000000000000**. In this case, the object UUID does not represent any object. Often, however, the object UUID represents a specific RPC object and is a non-nil value. To create a non-nil object UUID, a server calls the **uuid_create()** routine, which returns a non-nil UUID that the server then associates with a particular object.

If a client requests a non-nil object UUID, the client runtime uses that UUID as one of the criteria for a compatible server. When searching the namespace for server binding information, the client runtime looks for the requested interface identifier and object UUID. The endpoint map service uses this same information to help identify a map element corresponding to a compatible server.

Figure 12 illustrates the aspects of a server and its system that is identified by the client’s server binding information, requested interface identifier, and requested object UUID.

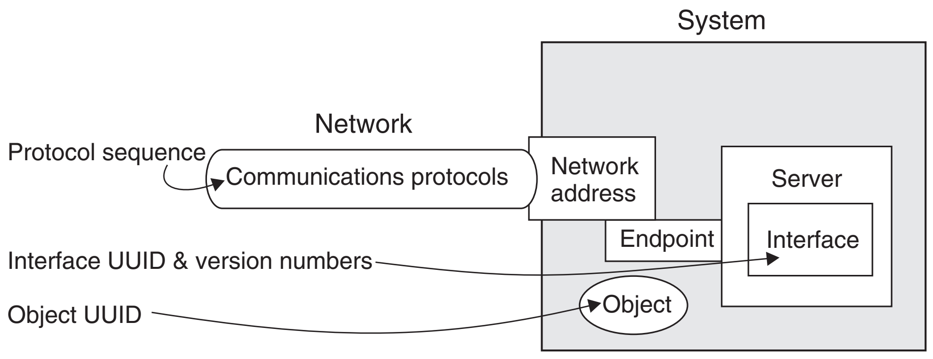


*Figure 12. Information Used to Identify a Compatible Server*

## How Clients Obtain Server Binding Information

When a client initiates a series of related remote procedure calls, the RPC runtime tries to establish a binding, which requires the address of a compatible server. An RPC client can use compatible binding information obtained from either a namespace or from a string representation of the binding information. Establishing a binding can also involve requesting an endpoint from the Endpoint Mapper of the server's system.

**Binding Information in a Namespace:** Usually, a server exports binding information for one or more of its interface identifiers and its object UUIDs, if any, to an entry in a namespace. A namespace is a collection of information about applications, systems, and other relevant computing resources which is maintained by a directory service, such as the Cell Directory Service (CDS). The namespace is provided by a directory service, such as the DCE Cell Directory Service (CDS). The directory service entry to which a server exports binding information is known as a **server entry**.

To learn about a server that offers a given RPC interface and object, if any, a client can import binding information from a server entry belonging to that server. A client can delegate the task of finding servers from the namespace to a stub. In this case, if a binding is accidentally broken, the RPC runtime automatically tries to establish a new binding with a compatible server.

Advantages of using a directory service to obtain binding information include:

- Convenient for large RPC environments. Initial overhead of understanding and configuring a directory service is balanced by easier management over time.
- Management of data in a directory service is more automated.
- Effective in dynamic end-user environments.

- Binding information is stored in a named server entry. Data can be dynamic. Servers can automatically place their binding information in the namespace. Changes in binding information are made once by a server or administrator and then propagated automatically by the directory service to the replicas of the data.
- Centralized administration of data in a namespace. Sophisticated access control is possible.
- Supports searching for and choosing services based on an interface identifier and object UUID. Clients access data by specifying an entry name. NSI groups and profiles in directory service entries provide search paths for importing binding information.

**Binding Information in Strings:** Occasionally, a client can receive binding information in the form of a string (also known as a string binding).

The client can receive a string binding (or the information to compose a string binding) from many sources; for example, an application-specific environment variable, a file, or the application user. The client must call the RPC runtime to convert a string binding to a binding handle. The RPC runtime stores the binding information from the string binding and creates a binding handle that refers to the binding information. The runtime returns this binding handle to the client to use for remote procedure calls.

String representations of binding information have several possible components. The binding information can include an RPC protocol sequence, a network address, and an endpoint. The protocol sequence is mandatory, the endpoint is optional. For a server on the client's system, the network address is optional. Also, a string binding optionally associates an object UUID with the binding information.

The string bindings have the following format:

*obj-uuid***@***rpc-protocol-seq***:***network-addr***[***endpoint,option-name=opt-value*...**]**

or

*obj-uuid***@***rpc-protocol-seq***:***network-addr***[endpoint**=*endpoint,option-name=opt-value...***]**

The following example string binding contains all possible components:

```
b07122e2-83df-11c9-be29-08002b1110fa@ncacn_ip_tcp:16.20.15.25[2001]
```

The following example string binding contains only the protocol sequence and network address:

```
ncacn_ip_tcp:16.20.15.25
```

For more information about this format, see the *z/OS DCE Application Development Reference*.

String bindings are useful in small environments, for example, when developing and testing an application. However, string bindings are inappropriate as the principal way of providing binding information to clients. For moderate to large environments and for small environments that may expand, applications should use the directory service to advertise binding information.

## Client Binding Information for Servers

When making a remote procedure call, the client runtime provides information about the client to the server runtime. This information, known as ***client binding information***, includes the following information:

- The address where the call originated (network address and endpoint)
- The RPC protocol used by the client for the call
- The object UUID that a client requests
- The client authentication information (if present).

The server runtime maintains the client binding information and makes it available to the server application by a ***client binding handle***.

Figure 13 illustrates the relationships between what a client supplies when establishing a binding and the corresponding client binding information.

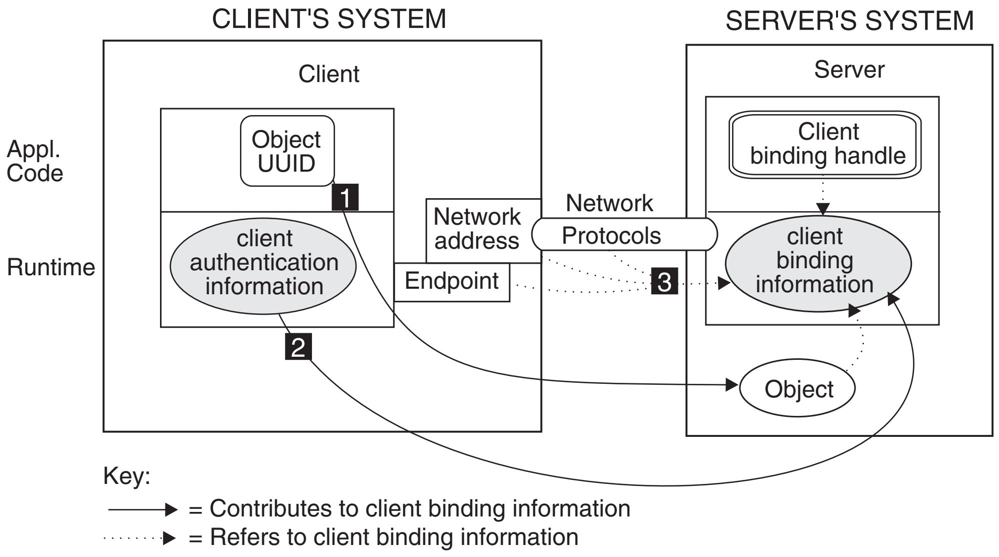


*Figure 13. Client Binding Information Resulting from a Remote Procedure Call*

The reference keys in Figure 13 refer to the following:

**1** The requested object UUID, which may be the nil UUID

**2** Client authentication information, which is optional

**3** The address from which the client is making the remote procedure call. The communications protocols supply this information to the server.

A server application can use the client binding handle to ask the RPC runtime about the object UUID requested by a client or about the client’s authentication information.

## Endpoints

An endpoint is the address of a specific server instance on a host system. The following kinds of endpoints exist: well-known endpoints and dynamic endpoints.

### Well-Known Endpoints

A ***well-known endpoint*** is a preassigned stable address that a server can use every time it runs. Well-known endpoints typically are assigned by a central authority responsible for a transport protocol; for example, the ARPANET Network Information Center assigns endpoint values for the IP family of protocols. If you use well-known endpoints for a server, you should register them with the appropriate authority.

Well-known endpoints can be declared for an interface (in the interface declaration) or for a server instance, as follows:

- Any interface definition can store one or more endpoints, along with the RPC protocol sequence corresponding to each endpoint (the endpoint attribute).

  When compiling an interface, the IDL compiler stores each combination of endpoint and protocol sequence in the interface specification. If a call is made using binding information that lacks an endpoint, the RPC runtime automatically looks in the interface specification for a well-known endpoint specified for the protocol sequence obtained from the binding information. If the interface specification contains an appropriate endpoint, the runtime adds it to the binding information.
- Alternatively, server-specific, well-known endpoints can be defined in server application code. When asking the runtime to use a given protocol sequence, the server supplies the corresponding endpoints to the RPC runtime. On a given system, each endpoint can be used by only one server at a time. If server application code contains a hardcoded endpoint or the server's installers always specify the same well-known endpoint, only one instance of the server can run per system.

When a server exports its binding information to a server entry, the export operation includes any well-known endpoints within the server binding information stored in the server entry.

### Dynamic Endpoints

A ***dynamic endpoint*** is an endpoint that is requested and assigned at runtime. For some transport protocols, the number of endpoints is limited; for example, TCP/IP and UDP/IP use a 16-bit number for endpoints, which allows only 65,536 endpoints. When the supply of endpoints for a transport protocol is limited, the protocol ensures an adequate supply of endpoints by limiting the portion that can be reserved as well-known endpoints. A transport, on request, dynamically makes its remaining endpoints available on a first-come, first-served basis to specific processes such as RPC server instances.

When a server requests dynamic endpoints, the server's RPC runtime asks the operating system for a unique dynamic endpoint for each protocol sequence the server is using. For a given protocol sequence, the local implementation of the corresponding transport protocol provides the requested endpoints. When an RPC server with dynamic endpoints stops running, its dynamic endpoints are released.

Because of the transient nature of dynamic endpoints, the NSI does not export them to a namespace; however, the NSI does export the rest of the server's binding information. References to expired endpoints would remain indefinitely in server entries, causing clients to import and try, unsuccessfully, to establish bindings to nonexistent endpoints. Also, updating transient data in namespace entries impairs the performance of a directory service. Therefore, the export operation removes dynamic endpoints before adding binding information to a server entry; the exported server address contains only network addressing information. The import operation returns a partially bound binding handle. The client makes its first remote procedure call with the partially bound handle, and the endpoint map service on the server's system attempts to resolve the binding handle with the endpoint of a compatible server. To make dynamic endpoints available to clients using partially bound binding handles, a server must register its dynamic endpoints in the local endpoint map.

**Note:** Register all endpoints to enable administrators to view all the endpoints of RPC servers by showing the endpoint map elements. To do this, use the **dcecp endpoint show** operation of the DCE control program.

By using object UUIDs, a server can ensure that a client that imports a partially bound handle obtains one of a particular server's endpoints. The server must do the following:

1. Specify a list of one or more object UUIDs that are unique to the server.
2. Export the list of object UUIDs.
3. Supply the list of object UUIDs to the endpoint map service when registering endpoints.
4. If the server provides different managers that use an interface for different types of objects, the server must set the type of each object.

To request binding information for a particular server, a client specifies one of the server's object UUIDs, which is then associated with the server binding information the client uses for making a remote procedure call.

**Note:** If a client requests the nil object UUID when importing from a server entry containing object UUIDs, the import (or lookup) operation selects one of those object UUIDs and associates it with the imported server binding information. This object UUID ensures that the call goes to the server that exported the binding information and object UUID to the server entry.

## Run time Semantics

Run time semantics identify the ability of a procedure to run more than once during a given remote procedure call. The communications environment that underlies remote procedure calls affects the reliability of the RPC. A communications link can break for a variety of reasons, such as a server stopping, a remote system failure, a network failure, and so forth; all remote procedures risk disruption because of communications failures. However, some procedures are more sensitive to such failures, and their affect depends partly on how reinvoking an operation affects that operation's results.

To maximize valid outcomes for its operations, the operation declarations of an RPC interface definition indicate the effect of running operations multiple times on the outcome of the operations.

The run time semantics for DCE RPC calls are summarized in Table 3.

*Table 3 (Page 1 of 2). Run time Semantics*

| Semantics | Meaning |
|---|---|
| ***At-most-once*** | The operation must run either once, partially, or not at all; for example, adding or deleting an appointment from a calendar can use at-most-once semantics. This is the default run time semantics for remote procedure calls. |

*Table 3 (Page 2 of 2). Run time Semantics*

| Semantics | Meaning |
|---|---|
| ***Idempotent*** | The operation can run more than once; running more than once using the same input arguments produces identical outcomes without undesirable side effects; for example, an operation that reads a block of an immutable file is **idempotent**. DCE RPC supports ***maybe semantics*** and ***broadcast semantics*** as special forms of **idempotent** operations. |

| Semantics | Meaning |
|---|---|
| Maybe | The caller neither requires nor receives any response or fault indication for an operation, even though the operation may not have completed. An operation with **maybe** semantics is implicitly idempotent and must lack output parameters. |
| Broadcast | The operation is always broadcast to all host systems on the local network, rather than delivered to a specific system. An operation with **broadcast** semantics is implicitly **idempotent**. |

The broadcast capabilities of RPC runtime have a number of distinct limitations:

- Not all systems and networks support broadcasting. In particular, broadcasting is not supported by the RPC connection-oriented protocol.
- Broadcasts are limited to hosts on the local network.
- Broadcasts make inefficient use of network bandwidth and processor cycles.
- The RPC runtime library does not support **at-most-once** semantics for broadcast operations; it applies **idempotent** semantics to all such operations.
- The input arguments for broadcast calls are limited to 944 bytes.

**Notes:**

1. Authenticated RPC for **maybe** calls is not supported on z/OS DCE.
2. It does not matter to a DCE application whether UDP or TCP is used, when a data frame is discarded because of network congestion. A client application will not receive an error because a retransmission occurs whether TCP or UDP is used.

   If both the client and server systems remain running but are susceptible to lost UDP frames, the RPC runtime does guarantee the call exactly once, not just **at-most-once**. For the normal non-idempotent situation, the RPC runtime keeps state information including a sequence number on each request. The server RPC runtime can therefore detect whether this is a new situation where the frame is lost going to the server, or a duplicate situation where the frame is lost coming from a server. In effect, the RPC runtime has implemented a reliable connection-oriented protocol over a connectionless UDP network.

   If it is known that the server can run multiple times without any adverse affects, specifying **idempotent** removes the requirement of the RPC runtime to keep sequence numbers. The runtime resends the request until the server answers and does not worry about the server running more than once.

   If either the client or server stops running in the middle of an RPC, the results of that RPC are indeterminate as two-phase commit is unavailable. That is, transactional RPC is not supported by z/OS DCE base services. (However Application Support does support transactional RPC.)

## Communications Failures

If a server's runtime detects a communications failure during a remote procedure call, the server's runtime attempts to end the now orphaned call by sending a cancel to the called procedure. A **cancel** is a mechanism by which a running client thread notifies a running server thread (the canceled thread) to end as soon as possible. A cancel sent by the RPC runtime after a communications failure initiates orderly ending of an RPC. (For a brief discussion of how cancels work with remote procedure calls, see Chapter 7, "RPC and Other DCE Components" on page 85. For detailed information, see "Thread Cancelation" on page 328.)

Applications that use context handles to establish a client context require a context rundown procedure to enable the server to clean up the client context when it is no longer needed. A type declaration for the context rundown procedure is declared in the RPC interface definition; this ensures that the stub knows about the rundown procedure in the server application code. If a communications link with a client is lost while a server is maintaining context for the client, the RPC runtime will tell the server to start the context rundown procedure. This is a user-written procedure. See "The Context Rundown Procedure" on page 270 for more information on specifying a context rundown procedure, and see Figure 54 on page 273 for an example.

## Scalability

Unlike local applications, RPC applications require network resources, which are possible bottlenecks to scaling an RPC application. RPC clients and servers require network resources that are not required by local programs. The main *network resources* to consider are network bandwidth, endpoints, *network descriptors* (the identifiers of potential network channels such as UNIX sockets), kernel buffers, and for a connection-oriented transport, the connections. Also, RPC applications place extra demands on system resources such as memory buffers, various quotas, and the processing unit.

The number of remote procedure calls that a server can support depends on various factors, such as the following:

- The resources of the server and the network
- The requirements of each call
- The number of calls that can be concurrently offered at some level of service
- The performance requirements.

An accurate analysis of the requirements of a given server involves detailed work load and resource characterization and modeling techniques. Although measurement of live configurations under load will offer the best information, general guidelines apply. You should consider the connection, buffering, bandwidth, and processing unit resources as the most likely RPC bottlenecks to scaling. Use these application requirements to scale resources.

Many system implementations limit the number of network connections per process. This limit provides an upper bound on the number of clients that can be served concurrently using the connection-oriented protocol. The limit on z/OS DCE is 64. However, except for applications that use context handles, the connection-oriented RPC runtime allows pooling of connections. Pooling permits simultaneous support of more clients than the maximum number of connections, provided they do not all make calls at the same instant and occasionally can wait briefly.

## RPC Objects

An RPC object is an entity that an RPC server defines and identifies to its clients. Frequently, an RPC object is a distinct computing resource such as a particular database, directory, device, process, or processor. Identifying a resource as an RPC object enables an application to ensure that clients can use an RPC interface to operate on that resource. An RPC object can also be an abstraction that is meaningful to an application, such as a service or the location of a server.

The RPC runtime provides substantial flexibility to applications about whether, when, and how they use RPC objects. RPC applications generally use RPC objects to enable clients to find and access a specific server. When servers are completely interchangeable, using RPC objects may be unnecessary. However, when clients need to distinguish between two servers that offer the same RPC interface, RPC objects are essential. If the servers offer distinct computing resources, each server can identify itself by defining its resources as RPC objects. Alternatively, each server can establish itself as an RPC object that is distinct from other instances of a server offering the same RPC interface.

RPC objects also enable a single server to distinguish among alternative implementations of an RPC interface, as long as each implementation operates on a distinct type of object. To offer multiple implementations of an RPC interface, a server must identify RPC objects, classify them into types, and associate each type with a specific implementation.

The set of remote procedures that carries out an RPC interface for a given type of object is known as a ***manager***. The tasks performed by a manager depend on the type of object on which the manager operates. For example, a manager of a queue-management interface may operate on print queues, while another manager may operate on batch queues.

# Chapter 6. Basic RPC Routine Usage

This chapter introduces a number of basic DCE RPC routines for directory service, communications, and authentication operations and discusses major usage issues important for developing DCE RPC applications.

This chapter discusses the following topics:

- Overview of the basic RPC runtime routines
- Server initialization tasks
- How clients find servers

## Overview of the Basic RPC Runtime Routines

This section summarizes the major concerns of RPC communications, Name Service Interface (NSI) usage, and authenticated RPC.

- Basic operations of RPC communications

  The DCE RPC runtime provides the following communications operations for RPC applications:

  – Managing communications for RPC applications

    As part of server initialization, a server sets up its communications capabilities by a series of calls to the RPC runtime. These runtime calls register the server's RPC interfaces, tell the RPC runtime what combination of communications protocols to use for the server, and register the endpoints of the server for each of its interfaces. After completing these and any other initialization tasks, the server tells the runtime to begin listening for incoming calls.

  – Managing binding information

    A variety of communications operations allow servers to access and manipulate binding information. In addition, a set of communications operations enables applications to manipulate string representations of binding information (string bindings).

- Basic operations of the RPC NSI

  The NSI routines perform operations on a namespace for RPC applications. The fundamental operations are:

  – Creating and deleting entries in namespaces

  – Exporting

    A server uses the NSI export operation to place binding information associated with its RPC interfaces and objects into the namespace used by the RPC application.

– Importing

  Clients can search for exported binding information associated with an interface and object using the NSI *import operation* or *lookup operation*. These two operations are collectively referred to as the NSI *search operations*.

– Unexporting

  The unexport operation enables a server to remove some or all of its binding information from a server entry.

– Managing information in a namespace

  Applications use the NSI interface to place information about server entries into a namespace and to inquire about and manage that information.

- Basic operations of authenticated RPC

  The authenticated RPC routines provide a mechanism for establishing secure communications between clients and servers.

  To engage in authenticated RPC, a client and server must agree on the authentication service to be used. The server's responsibility is to register its principal name and the authentication service to be supported with the RPC runtime. The client's responsibility is to establish the authentication service, a given protection level, and an authorization service for the server binding handle. The protection level determines the degree of protection applied to individual messages between the client and server. The authorization service determines the form in which the client's credentials will be presented to the server (for access checking).

  Once authenticated RPC has been established between a client and server, the client issues RPCs in the usual fashion, with all authentication and protection being handled by the DCE Security Service component and the RPC runtime.

Table 4 relates several of the RPC runtime operations just discussed to specific routines or sets of routines.

*Table 4 (Page 1 of 2). Basic Runtime Routines*

| Description of Operation | Usage | Routine Name(s) |
|---|---|---|
| **Communications Routines**: | | |
| Setting the type of an RPC object with the RPC runtime | Server | **rpc_object_set_type()** |
| Registering RPC interfaces | Server | **rpc_server_register_if()** |
| Selecting RPC protocol sequences | Server | **rpc_network_inq_protseqs()**<br>**rpc_server_use_all_protseqs()**<br>**rpc_server_use_all_protseqs_if()**<br>**rpc_server_use_protseq()**<br>**rpc_server_use_protseq_ep()**<br>**rpc_server_use_protseq_if()** |
| Obtaining server binding handles | Server | **rpc_server_inq_bindings()** |
| Registering endpoints | Server | **rpc_ep_register()**<br>**rpc_ep_register_no_replace()** |
| Unregistering endpoints | Server | **rpc_ep_unregister()** |
| Resolving endpoints | Client | **rpc_ep_resolve_binding()** |
| Listening for calls | Server | **rpc_server_listen()** |

*Table 4 (Page 2 of 2). Basic Runtime Routines*

| Description of Operation | Usage | Routine Name(s) |
|---|---|---|
| Manipulating string representations of binding information (string bindings) | Client | **rpc_binding_from_string_binding()** |
| | Client, Server | **rpc_binding_to_string_binding()** **rpc_string binding_compose()** **rpc_string_binding_parse()** |
| Changing the RPC object in server binding information | Client | **rpc_binding_set_object()** |
| Converting a client binding handle to a server binding handle | Server | **rpc_binding_server_from_client()** |
| **Name Service Interface Routines**: | | |
| Exporting binding information to a namespace | Server | **rpc_ns_binding_export()** |
| Searching a namespace for binding information | Client | **rpc_ns_binding_import_begin()** **rpc_ns_binding_import_done()** **rpc_ns_binding_import_next()** **rpc_ns_binding_lookup_begin()** **rpc_ns_binding_lookup_done()** **rpc_ns_binding_lookup_next()** **rpc_ns_binding_select()** |
| **Authentication Routines**: | | |
| Authentication and authorization | Server, Client | **rpc_binding_inq_auth_info()** **rpc_mgmt_set_authorization_fn()** |
| | Server | **rpc_binding_inq_auth_caller()** **rpc_binding_inq_auth_client()** **rpc_server_register_auth_info()** |
| | Client | **rpc_binding_set_auth_info()** |

## Server Initialization Using the RPC Routines

Before a server can receive any RPCs, it usually initializes itself by calling the **dce_server_register()** routine so that the server is properly recognized by DCE. However, servers can instead use a series of the lower-level RPC runtime routines. The server initialization code, that you write, varies among servers. However, every server must set up its communications capabilities, which usually involve most of the following tasks:

1. Assigning types to objects
2. Registering at least one interface
3. Specifying which protocol sequences the server will use
4. Obtaining a list of references to a server's binding information (a list of binding handles)
5. Registering endpoints
6. Exporting binding information to a server entry or entries in the namespace
7. Listening for RPCs
8. Performing cleanup tasks including unregistering endpoints

The following pseudocode illustrates the calls a server makes to accomplish these basic initialization tasks.

```
/* Initialization tasks */

   rpc_object_set_type(...);

   rpc_server_register_if(...);

   rpc_server_use_all_protseqs(...);

   rpc_server_inq_bindings(...);

   rpc_ep_register(...);

   rpc_ns_binding_export(...);

   rpc_server_listen(...);


/* Cleanup tasks */

   rpc_ep_unregister(...);
```

## Assigning Types to Objects

An *object type* is a mechanism for associating a set of RPC objects and the manager whose remote procedures constitute an RPC interface for those objects. Using types, an application can cluster objects such as computing resources according to any relevant criteria. For example, a single accounting interface can be coded to operate on accounting databases that contain equivalent information but that are formatted differently; each database format represents a distinct type.

To simultaneously offer alternative implementations of an RPC interface for different types of objects, a server uses alternative managers. Servers that have each of their interfaces on only one manager can usually avoid the tasks associated with assigning object types. However, when a server offers multiple managers, each manager must be dedicated to operating on a separate type of object. In this case, a server must classify some or all of its objects into types; for example, a calendar application that specifies one non-nil type UUID for departmental calendars and another non-nil type UUID for personal calendars.

By default, objects have the nil type. Only a server that uses different managers for different objects or sets of objects needs to type classify its RPC objects. To type classify an object, a server associates the object UUID of the object with a single ***type UUID*** by calling the **rpc_object_set_type()** procedure separately for each object. To create a type UUID, a server calls the **uuid_create()** routine.

The exact implementation of a manager can vary with the type of object on which each manager operates. For example, a queue-management interface may be implemented to manage print queues as objects in one case and to manage batch queues as objects in another. Figure 14 on page 77 illustrates the use of type UUIDs to identify two types of managers. The server associates each object with a particular type.

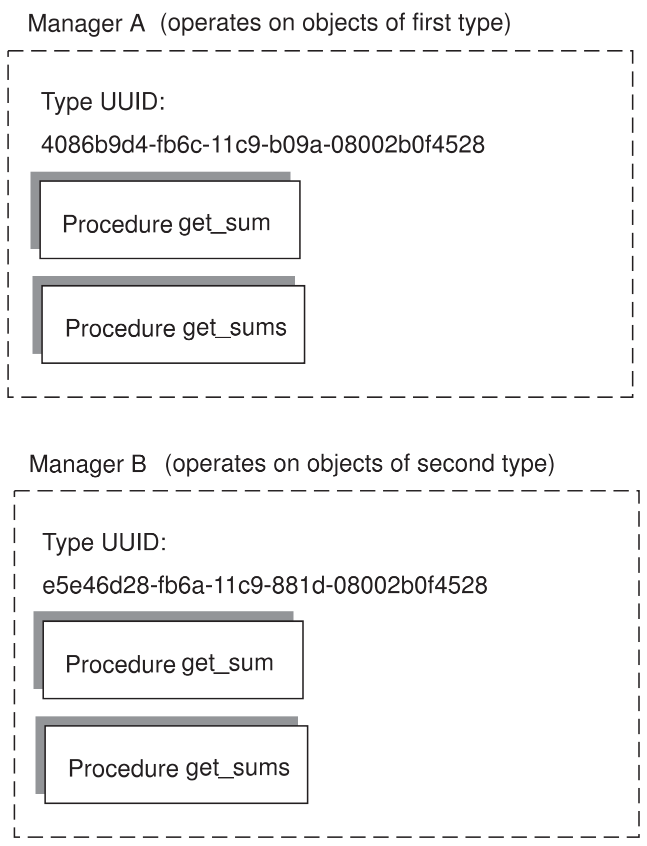

Manager A (operates on objects of first type)
Type UUID:
4086b9d4-fb6c-11c9-b09a-08002b0f4528
Procedure get_sum
Procedure get_sums
Manager B (operates on objects of second type)
Type UUID:
e5e46d28-fb6a-11c9-881d-08002b0f4528
Procedure get_sum
Procedure get_sums

*Figure 14. Types of Managers*

When the server receives an incoming call that specifies an object UUID, the server sends the call to the manager for the type to which the object belongs. For information on how a type is used to select a manager for an incoming call, see Chapter 10, “Topics in RPC Application Development” on page 173.

## Registering Interfaces

A server calls the **rpc_server_register_if()** routine to tell the RPC runtime about a specific RPC interface. Registering an interface informs the runtime that the server is offering that interface and makes it available to clients. A server can register any number of interfaces with the RPC runtime by calling the **rpc_server_register_if()** routine once for each set of procedures, or manager, that constitutes an interface.

To offer more than one manager for an interface, a server must register each manager separately.

When registering an interface, the server provides the following information:

- Interface specification

  This is a reference to information about an RPC interface as offered by its server stub. The DCE IDL compiler generates an interface specification as part of the stub code. For a specific version of an interface, all managers use the same ***interface specification***. Information in an interface specification that concerns application developers includes the following:

  – The interface identifier (UUID and major and minor version numbers).

  – The supported transfer syntaxes.

  - A list of any well-known endpoints (and their associated protocol sequences) specified in the interface definition (IDL) file.
  - The default manager endpoint vector (manager EPV) of the interface, (if present)

    A default manager EPV, constructed using the operation names of the interface definition, is typically generated for stubs by the DCE IDL compiler (the **no_mepv** compiler option suppresses this feature).

- A type UUID for the manager

  Each implementation of an interface, a manager, is represented by a type UUID.

- A manager EPV for the interface

  A server can register a given interface more than once by specifying a different type UUID and manager EPV each time it calls **rpc_server_register_if()**.

  A manager EPV is a list of the addresses (the entry points of the remote procedures provided by the manager) that represent the location in the code for each remote procedure implementation. A manager EPV must contain exactly one entry point for each procedure defined in the interface definition.

  The server can use the default manager EPV only once, and only for a manager that uses the procedure names declared in the interface definition. For any additional manager of the RPC interface, the server must create and register a unique manager EPV. Also, each manager must be associated with a distinct type UUID.

## Selecting RPC Protocol Sequences

A server can inquire about whether the local RPC runtime supports a specific protocol sequence by using the **rpc_network_is_protseq_valid()** routine. The server can also use the **rpc_network_inq_protseqs()** routine to ask the RPC runtime for a list of all protocol sequences supported by both the RPC runtime and the operating system.

To prepare to receive RPCs, a server uses **rpc_server_use_all_protseqs()** or **rpc_server_use_protseq()** calls to tell the RPC runtime to use at least one protocol sequence. For each protocol combination, the RPC runtime creates one or more binding handles with dynamic endpoints on which the server will listen for RPCs. The server then can use a list of these binding handles to register dynamic endpoints in the endpoint map and to export its binding information (except the endpoints) to the name service.

Optionally, an interface can contain one or more well-known endpoints, each of which is accompanied by a protocol sequence. A server uses the **rpc_server_use_all_protseqs_if()**, **rpc_server_use_protseq_if()**, or **rpc_server_use_protseq_ep()** to notify the RPC runtime about which protocol sequence and well-known endpoint combinations will be used.

A server can use any protocol sequence declared in an interface endpoint declaration, or the server can ignore the endpoint declarations, as long as it registers at least one endpoint.

## Obtaining a List of Server Binding Handles

After a server passes to the RPC runtime the protocol sequences over which it will listen for RPCs, the RPC runtime constructs server binding handles. Each binding handle refers to a complement of binding information that defines one potential binding, that is, a specific RPC protocol sequence, RPC protocol major version, network address, endpoint, and transfer syntax that an RPC client can use to establish a binding with an RPC server.

Before registering endpoints or exporting binding information, a server must obtain a list of its binding handles from the RPC runtime by using the **rpc_server_inq_bindings()** routine. The server passes this list back to the runtime as an argument when registering endpoints and exporting binding information.

## Registering Endpoints

Servers can use well-known or dynamic endpoints with any protocol sequence.

When a server asks the runtime to use a dynamic endpoint with a protocol sequence, the runtime asks the operating system to generate the endpoint. To use the dynamic endpoints, a server must register the server's binding information, including the endpoints, by using the **rpc_server_ep_register()** routine. For each combination of RPC interface identifier, object UUID, and binding information that the server offers, the endpoint map service creates an element in the local endpoint map.

A server does not necessarily need to register well-known endpoints; however, by registering well-known endpoints, the server ensures that clients can always obtain them. Registration also makes the endpoints accessible to administrators, who can show the elements of an endpoint map by using the **endpoint show** operation of the DCE control program **dcecp**.

Servers can remove map elements from a local endpoint map using the **rpc_ep_unregister()** routine. Servers should unregister endpoints after they stop listening.

## Making Binding Information Accessible to Clients

A server needs to make its binding information accessible to clients. Usually, a server uses the NSI export operation to place its binding information into a server entry. However, servers can also make string bindings accessible to clients. In any case, the server obtains its binding information from the runtime by first using the **rpc_server_inq_bindings()** routine to ask for a list of binding handles.

**Using String Bindings to Provide Binding Information:** While running and debugging a server program you may temporarily want to communicate binding information to clients using string bindings. A server can then establish a client/server relationship without registering endpoints in the local endpoint map or exporting binding information to a namespace.

The server can convert into a string each binding handle in the list obtained from the **rpc_server_inq_bindings()** routine, by calling **rpc_binding_to_string_binding()**. The resulting string binding is always fully bound. The server then makes some or all of its string bindings available to clients in various ways; for example, by placing the string bindings in a file to be read by clients or users or both.

**Exporting Binding Information:** Servers can export binding information (and interface identifiers) or objects or both by calling the **rpc_ns_binding_export()** routine. To export binding information associated with a given RPC interface, a server uses an ***interface handle***. The interface handle is created by the IDL compiler as a reference to information about the interface that the compiler stores in an interface specification. To refer to binding information, the application code obtains a list of server binding handles from the RPC runtime and passes the list to the export operation. The list contains binding handles for all the protocol sequence and endpoint combinations that the server has requested; it obtains this list by calling the use-protocol-sequence operations. However, the server can remove any of those binding handles from the list before exporting it. This enables a server to export the binding information associated with a subset of its binding handles.

To export object UUIDs, a server application must provide a list of object UUIDs for the RPC objects it offers. The server can generate these object UUIDs itself or obtain them from some application-specific source such as an object-UUID database. All object UUIDs in a given server entry are associated with every interface UUID and server address in the entry.

Figure 15 illustrates the use of server binding handles to refer to server binding information to be exported.

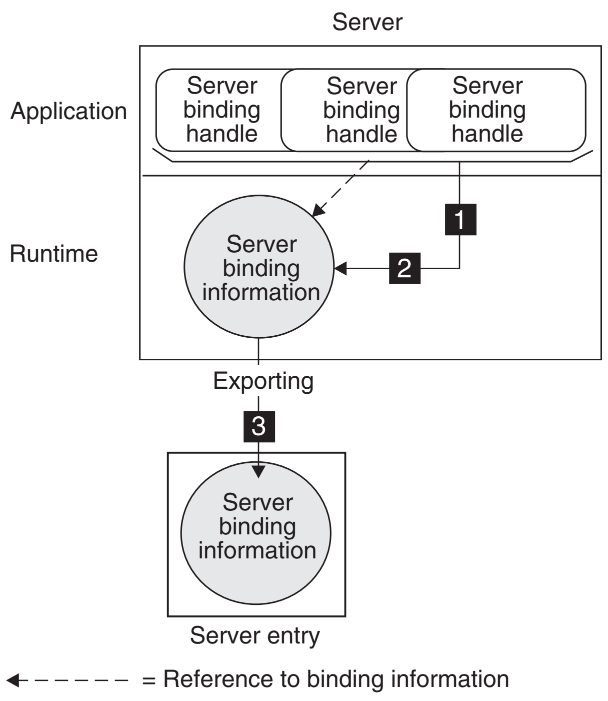


*Figure 15. Exporting Server Binding Information*

The reference keys in the figure refer to the following operations:

**1** The server application code calls the export operation, having previously inquired for a list of binding handles. Along with the name of a server entry, the application passes the export operation a list of server binding handles and an interface handle, a list of object UUIDs, or both.

**2** The export operation uses the binding handles to identify the binding information to export.

**3** The export operation places binding information, the associated interface identifier, and the associated list of object UUIDs into the designated server entry.

A server entry must belong exclusively to a server running on a given host. If there are identical, interchangeable instances of a server on the host, they can share a single set of server entries. However, if clients need to distinguish between coexisting instances of a server (for example, when each offers a different RPC object), each instance requires its own server entry.

**Note:** Cell Directory Services (CDS) databases are subject to access control. To access entries in a CDS database, you need ***Access Control List*** (***ACL***) permissions. Depending on the NSI operation, you need ACL permissions to the parent directory, or the CDS object entry, or both. If you need ACL permissions, see your CDS administrator. For more information on CDS, see the *z/OS DCE Application Development Guide: Directory Services*.

The ACL permissions are:

- To create an entry, you need insert permission to the parent directory.
- To read an entry, you need read permission to the CDS object entry.
- To write to an entry, you need write permission to the CDS object entry.
- To delete an entry, you need delete permission either to the CDS object entry or to the parent directory.
- To test an entry, you need either test permission or read permission to the CDS object entry.

Note that write permission does not imply read permission.

### Listening for Calls

When a server is ready to accept RPCs, it initiates listening, specifying the maximum number of calls it can run concurrently by calling the **rpc_server_listen** routine. If a server allows concurrent calls, its remote procedures are responsible for concurrency control (also known as *thread-safe*). If it runs a set of remote procedures concurrently requiring concurrency control and a server lacks this control, the server must allow only one call at a time.

The RPC runtime continues listening for new RPCs to the server's registered interfaces, until one of the following events occurs:

- Any of the server's procedures makes a local management call to stop a server from listening for future RPCs.
- For applications whose servers enable clients to stop servers from listening, a client makes a remote management call to stop a server from listening for future RPCs.

On receipt of a stop listening request, the RPC runtime stops accepting new RPCs for all registered interfaces. However, currently running calls are allowed to complete. After all running calls are completed, the listen operation stops listening and returns control to the server. Servers should unregister endpoints after they stop listening.

## How Clients Find Servers

A client runtime can obtain server binding information from a namespace. Alternatively, a client runtime can obtain server binding information in string format from an application-specific source, such as a file. Runtime routines enable client applications to obtain server binding handles that refer to server binding information obtained from either source.

## Searching a Namespace

To obtain binding information from a name service database, a client can do one of the following:

- The client uses the automatic method of binding management to make the client stub transparently manage binding information.

  In this case, the application code lacks any calls to the NSI interface. However, the automatic method does require the client to identify the directory service entry where the search for binding information begins. The client must specify the starting entry name as the value of the NSI-defined **RPC_DEFAULT_ENTRY** environment variable.
- The client calls the import routines **rpc_ns_binding_import_begin()**, **rpc_ns_binding_import_next()**, and **rpc_ns_binding_import_done()** to obtain a binding handle for a compatible server.
- The client calls the lookup routines **rpc_ns_binding_lookup_begin()**, **rpc_ns_binding_lookup_next()**, and **rpc_ns_binding_lookup_done()** to obtain a list of binding handles for a compatible server. Select a binding handle from the list by calling either of the following:
  - The NSI select routine **rpc_ns_binding_select()**, which selects a binding handle at random
  - A user-defined select routine, which runs an application-specific selection algorithm

An NSI import or lookup operation searches server entries for a compatible server. On finding such a server entry, the search operation copies the server binding information associated with the requested interface and an object UUID. The search operation then creates a randomly ordered list of server binding handles to refer to the potential bindings represented by the binding information.

Figure 16 illustrates the use of a server binding handle to refer to server binding information selected by an import operation.

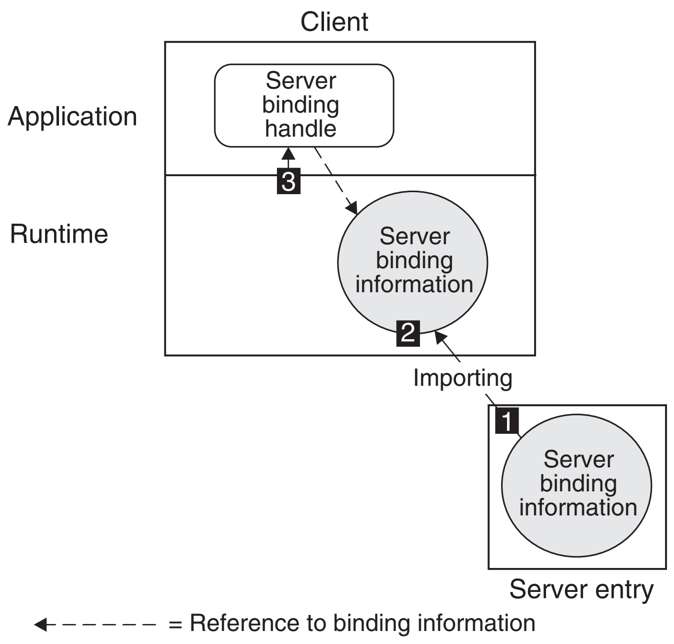

Client
Application
Runtime
Server binding handle
3
Server binding information
2
Importing
1
Server binding information
Server entry
= Reference to binding information

*Figure 16. Importing Server Binding Information*

The reference keys in the figure refer to the following operations:

**1** The import operation looks up binding information of a server that is compatible with the client.

The import operation finds a server entry based on the specified interface identifier, and then looks at the list of object UUIDs. If the importing client specifies a non-nil object UUID, the import operation looks for and returns that object UUID. If the client specifies the nil object UUID and the server entry contains any object UUIDs, the import operation selects and returns one UUID at random. If the entry lacks any object UUIDs, the import operation returns the nil UUID.

**2** The import operation fetches the compatible binding information and creates a binding handle for each potential binding represented in the binding information.

**3** The import operation then selects a binding handle at random and passes it to the client application.

**Note:** With z/OS DCE, any thread calling the RPC NSI routines (**rpc_ns_xx()**) cannot be canceled with the **pthread_cancel()** routine. This avoids potential resource clean up problems with the NSI routines.

## Using String Bindings to Obtain Binding Information

To use a string binding, a client starts with either an existing string binding or with the components of the binding information. Do *not* hardcode string bindings into application code. Rather, specify them at runtime using a command argument, environment variable, file, or other means. The simplest way to specify a string binding is for a user to supply a string binding manually to a client. However, this manual approach is awkward as users must know how to obtain and manipulate the string bindings. Also, if binding information changes, the users are responsible for updating any string bindings used by their clients. Reducing manual intervention in the use of string bindings requires that an application provide its own mechanisms for storing, maintaining, and accessing binding information. In contrast, a directory service such as CDS provides these mechanisms automatically to applications that store binding information in a namespace.

Regardless of how a client obtains a string binding, before establishing a binding, the client must ask the RPC runtime for a binding handle that refers to the server binding information depicted in the string binding. The client converts the string binding into a server binding handle by calling the **rpc_binding_from_string_binding()** routine.

The following pseudocode lists the calls for composing a string binding and for using it to obtain a server binding handle.

```
rpc_string_binding_compose(...);

rpc_binding_from_string_binding(...);
.
.
.
rpc_string_free(...);
```

# Chapter 7. RPC and Other DCE Components

This chapter discusses aspects of the internal behavior of RPCs that are significant for advanced RPC programmers, including the following topics:

- Threads in RPC applications
- Authenticated remote procedure calls
- Using the Name Service Interface (NSI)

DCE RPC is a fully integrated part of the distributed computing environment. The communications capabilities of DCE RPC are used by clients and servers of other DCE components. In turn, RPC uses services provided by the following DCE components: the Threads service, the Security Service, and the Cell Directory Service.

Thread services are also important to DCE RPC. A thread is a single sequential flow of control with one point of execution on a single processor at any instant. Multiple threads can coexist in a single process. DCE RPC uses threads internally for its own operations. DCE RPC also provides an environment where RPC applications can use thread services.

The DCE RPC runtime provides RPC applications with a programming interface to the DCE Security Service. The RPC authentication interface enables RPC clients and servers to mutually authenticate (that is, prove the identity of) each other. An authenticated RPC provides client authorization information and authentication information to servers. Authorization information includes the privileges a client has and the identities a client is associated with at the time of a call. By comparing client authorization information to access control lists, a server can find out whether a client is eligible to use a requested remote procedure. Client authentication information identifies a client to a server.

To help RPC clients find RPC servers, RPC applications typically use a namespace. A namespace is a collection of information about applications, systems, and any other relevant computing resources. A namespace is maintained by a directory service, such as the Cell Directory Service (CDS). DCE RPC provides a Name Service Interface (NSI) that is independent of any particular directory service.

NSI communicates with supported directory services for both RPC applications and the RPC control program. NSI insulates RPC applications from the intricacies of using a directory service. An RPC server uses NSI to store information about itself in a namespace, and a client uses NSI to access information about a server that meets the client's requirements for a specific RPC interface and object, among other things. The client uses this information to establish a relationship, known as a binding, with the server.

## Threads in RPC Applications

Each remote procedure call occurs in a run time context called a *thread*. A *thread* is a single sequential flow of control with one point at which it runs on a single processor at any instant. A thread created and managed by application code is an *application thread*.

Traditional processing occurs exclusively within *local application threads*. Local application threads run within the confines of one address space on a local system and pass control exclusively among local code segments, as illustrated in Figure 17 on page 86.

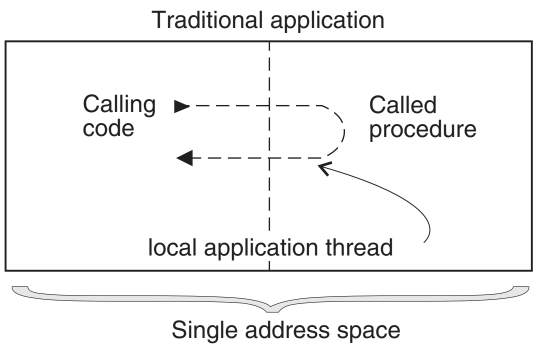


*Figure 17. Local Application Thread During a Procedure Call*

RPC applications also use application threads to issue both RPCs and runtime calls, as follows:

- An RPC client contains one or more *client application threads*; that is, a thread that runs client application code that makes one or more RPCs.
- A DCE RPC server contains one *server application thread*; that is, a thread that runs the server application code that listens for incoming calls.

In addition, in order to run called remote procedures, an RPC server uses one or more *call threads* provided by the RPC runtime. As part of initiating listening, the server application thread specifies the maximum number of concurrent calls it will run. The maximum number of call threads in multithreaded applications depends on the design of the application. The RPC runtime creates the same number of call threads in the server process. To determine the number of call threads created by the RPC runtime, see "Dynamic Executor Threads" on page 208.

The number of call threads is significant to application code. When using only one call thread, application code does not have to protect itself against concurrent resource use. When using more than one call thread, application code must protect itself against concurrent resource use.

Figure 18 shows a multithreaded server with a maximum of four concurrently running calls. Of the four call threads for the server, only one is currently in use; the other three threads are available for running calls.

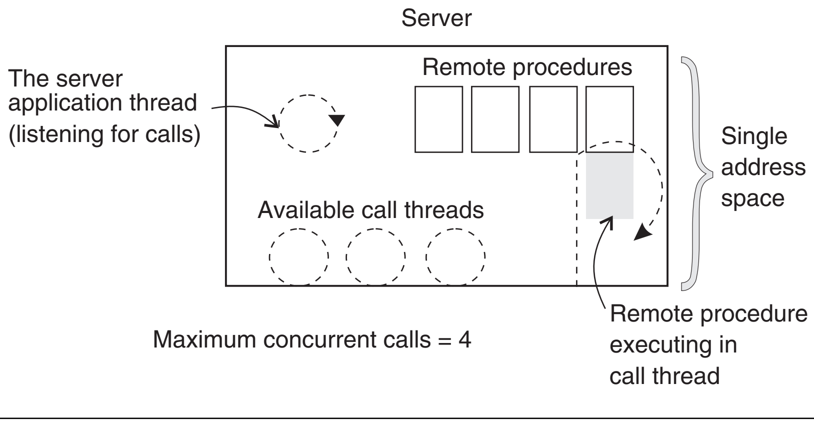

Server
Remote procedures
The server application thread (listening for calls)
Single address space
Available call threads
Maximum concurrent calls = 4
Remote procedure executing in call thread

*Figure 18. Server Application Thread and Multiple Call Threads*

## RPC Threads

In distributed processing, a call extends to and from client and server address spaces. Therefore, when a client application thread calls a remote procedure, it becomes part of a logical run time thread known as an *RPC thread*. An RPC thread is a logical construct that encompasses the various phases of a remote procedure call as it extends across actual threads and the network. After making an RPC, the calling client application thread becomes part of the RPC thread. Usually, the RPC thread maintains control until the call returns.

The RPC thread of a successful remote procedure call moves through the phases as illustrated in Figure 19.

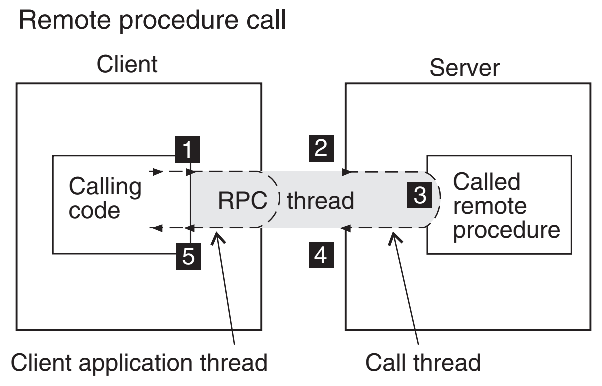


*Figure 19. Phases of an RPC Thread*

The phases of an RPC thread in the preceding figure include the following:

**1** The RPC thread begins in the client process, as a client application thread makes an RPC to its stub. At this point, the client thread becomes part of the RPC thread.

**2** The RPC thread extends across the network to the server address space.

**3** The RPC thread extends into a call thread, where the remote procedure runs.

While a called remote procedure is running, the call thread becomes part of the RPC thread. When the call finishes running, the call thread ceases being part of the RPC thread.

**4** The RPC thread then retracts across the network to the client.

**5** When the RPC thread arrives at the calling client application thread, the remote procedure call returns any call results and the client application thread ceases to be part of the RPC thread.

Figure 20 on page 88 shows a server running remote procedures in its two call threads, while the server application thread listens.

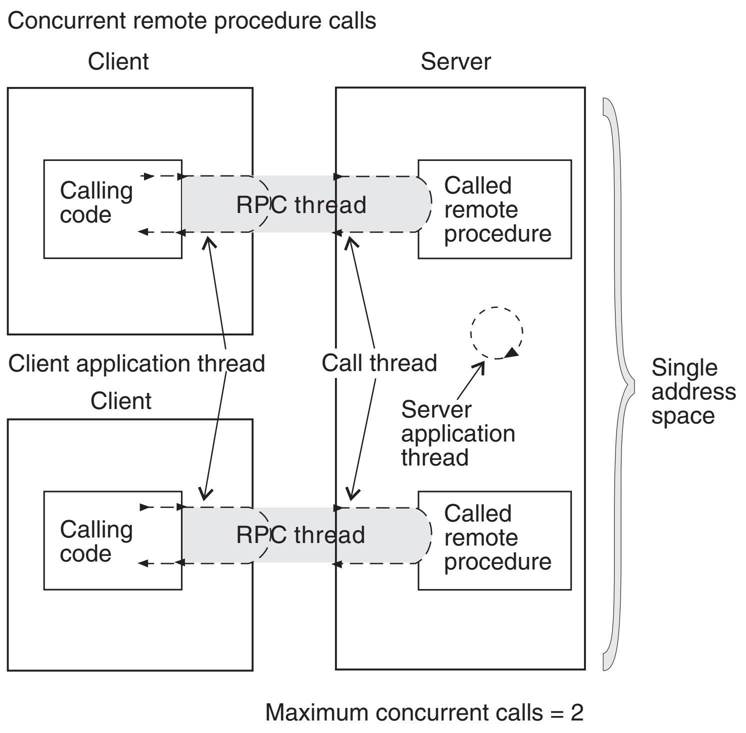


*Figure 20. Concurrent Call Threads Running in Shared Address Space*

**Note:** Although a remote procedure can be viewed logically as running within the exclusive control of an RPC thread, some parallel activity does occur in both the client and server.

An RPC server can concurrently run as many RPCs as it has call threads. When a server is using all of its call threads, the server application thread continues listening for incoming remote procedure calls. While waiting for a call thread to become available, DCE RPC server runtimes can queue incoming calls. Queuing incoming calls avoids RPCs failing during short-term congestion. The queue capacity for incoming calls is implementation dependent. Most implementations offer a small queue capacity. The queuing and routing of incoming calls is discussed in “Queuing Incoming Calls” on page 206.

## Cancel Operations

DCE RPC uses and supports the synchronous cancel capability provided by POSIX threads (**pthreads**). A *cancel* operation is a mechanism by which a thread informs another thread (the canceled thread) to end as soon as possible. Cancels operate on the RPC thread exactly as they would on a local thread, except for an application-specified, cancel-time-out period. A cancel-time-out period is an optional value that limits the amount of time the canceled RPC thread has before it releases control.

During an RPC, if its thread is canceled and the cancel-time-out period expires before the call returns, the calling thread regains control and the call is orphaned at the server. An orphaned call may continue to run in the call thread. However, the call thread is no longer part of the RPC thread, and the orphaned call cannot return results to the client.

A client application thread can cancel any other client application thread in the same process (it is possible, but unlikely, for a thread to cancel itself). While running as part of an RPC thread, a call thread can be canceled only by a client application thread.

A cancel operation goes through several phases. Figure 21 on page 89 indicates the point in the RPC thread where each of these phases occurs.

.

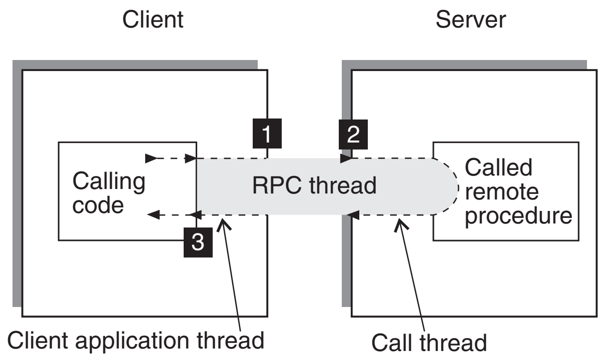


*Figure 21. Phases of a Cancel in an RPC Thread*

The phases of a cancel in Figure 21 include the following:

**1** A cancel that becomes pending at the client application thread at the start of or during an RPC becomes pending for the entire RPC thread. Thus, while still part of the RPC thread, the call thread also has this cancel pending.

**2** If the call thread of an RPC thread makes a cancelable call when cancels are not deferred and a cancel is pending, the cancel exception is raised.

**3** The RPC thread returns to the canceled client application thread with one of the following outcomes:

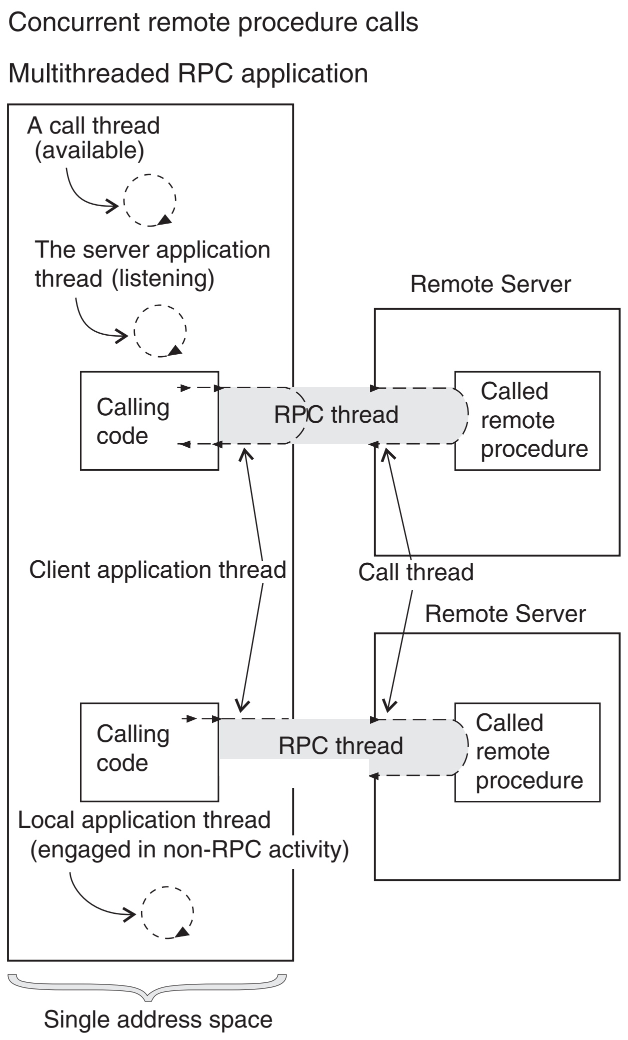
- If a cancel exception has not been taken, the RPC thread returns normal call results (output arguments, return value, or both) with a pending cancel.
- If the remote procedure is using an exception handler, a cancel exception can be handled. The procedure resumes, and the RPC thread returns normal call results without pending any cancel. (For information on the use of exception handlers, see Chapter 16, “Using the DCE Threads Exception-Returning Interface” on page 337.)
- If the remote procedure failed to handle a raised cancel exception, the RPC thread returns with the cancel exception still raised. This is returned as a fault.
- If the cancel-time-out period expires, the RPC thread returns either a cancel-time-out exception or status code, depending on how the application sets up its error handling. This is true for all cases where any abnormal ending is returned.

## Multithreaded RPC Applications

DCE RPC provides an environment for RPC applications that create multiple application threads (*multithreaded applications*). The application threads of a multithreaded application share a common address space and much of the common environment. If a multithreaded application must be *thread-safe* (ensuring that multiple threads can run simultaneously and correctly), the application is responsible for its own concurrency control. Concurrency control involves programming techniques, such as controlling access to code that can share a data structure or other resource, to prevent conflicting access by separate threads.

A multithreaded RPC application can have diverse activities going on simultaneously. A multithreaded client can make concurrent RPCs, and a multithreaded server can handle concurrent RPCs. Using multiple threads allows an RPC client or server to support local application threads that continue processing independently of RPCs.  Also, multithreading enables the server application thread and the client application threads of an RPC application to share a single address space as a joint *client/server instance*.  A multithreaded RPC application can also create local application threads that are not involved in the RPC activity of the application.

Figure 22 shows an address space where application threads are running concurrently.



Concurrent remote procedure calls
Multithreaded RPC application
A call thread (available)
The server application thread (listening)
Remote Server
Calling code
RPC thread
Called remote procedure
Client application thread
Call thread
Remote Server
Calling code
RPC thread
Called remote procedure
Local application thread (engaged in non-RPC activity)
Single address space

*Figure 22. A Multithreaded RPC Application Acting as Both Server and Client*

The application threads in Figure 22 are performing the following activities:

- The server application thread is listening for calls.
- A call thread is available to run an incoming remote procedure call.
- One client application thread has separated from an RPC thread and another is currently part of an RPC thread.
- A local application thread is engaging in non-RPC activity.

## Security and RPC: Using Authenticated RPC

DCE RPC supports authenticated communications between clients and servers. Authenticated RPC works with the authentication and authorization services provided by the DCE Security Service.

On the application level, a server makes itself available for authenticated communications by registering its principal name and the authentication service that it supports with the RPC runtime. The server principal name is the name used to identify the server as a principal to the Registry Service provided by DCE Security Service. In practice, this name is usually the same as the name that the server uses to register itself with the DCE Directory Service.

A client must establish the authentication service, protection level, and authorization service that it wants to use in its communications with a server. The client identifies the intended server by means of the principal name that the server has registered with the RPC runtime. Once the required authentication, protection, and authorization parameters have been established for the server binding handle, the client issues RPCs to the server as it normally does.

The DCE Security Service, in conjunction with the RPC runtime, assumes responsibility for the following:

- Authenticating the client and server in accordance with the requested authentication service
- Applying the requested level of protection to communications between the client and server
- Providing client authorization data to the server in a form determined by the requested authorization service.

**Note:** For a detailed discussion of authenticated RPC within the context of DCE Security, refer to Part 5, "Using the DCE Security APIs" on page 395.

### Authentication

When a client establishes authenticated RPC, it must indicate the authentication service that it wants to use. The possible values are the following:

| | |
|---|---|
| **rpc_c_authn_none** | No authentication |
| **rpc_c_authn_dce_secret** | DCE shared-secret key authentication |
| **rpc_c_authn_dce_public** | DCE public key authentication (reserved for future use) |
| **rpc_c_authn_default** | DCE default authentication service |

**Note:** Only the **dce_secret** and **none** authentication services are supported by z/OS DCE.

The value **rpc_c_authn_none** is used to turn off authentication already established for a binding handle. The default authentication is DCE shared-secret authentication, which is described in detail in Part 5, "Using the DCE Security APIs" on page 395.

Before a client and server can engage in authenticated RPC, they must *agree* on which authentication service to use. Specifically, the server must register the *agreed on* authentication service with the RPC runtime, along with the server's principal name. For its part, the client must select the same service for the server's binding handle. The client indicates the appropriate server by supplying the server's principal name. If the client does not know the server's name, it can use the **rpc_mgmt_inq_server_princ_name()** routine to determine the name. The actual RPC routines used by both the client and the server to establish authenticated RPC are described in "Authenticated RPC Routines" on page 94.

**Note:** All application servers should log in to DCE on their own using the **sec_login_** APIs. Otherwise, when starting multiple application servers which inherit the same login context, you need to either stagger the initiation of each server or initiate a single server prior to initiating the other servers. Otherwise you may receive a **sec_rgy_server_unavailable** status from any authentication RPC call. There is no problem for a single application server inheriting a login instance from a job, for example, if you log in to DCE prior to starting a single instance of an application server.

**Cross-Cell Authentication:** A client can engage in authenticated RPC with a target server that is in the client's cell or in a foreign cell. In the case of cross-cell authentication, DCE Security performs the necessary additional steps on behalf of the client.

To establish authenticated RPC with a foreign server, a client must supply the fully qualified principal name of the server. A fully qualified name includes the name of the cell as well as the name of the principal and takes the following form:

/.../*cell_name*/*principal_name*

**Protection Levels:** When a client establishes authenticated RPC, it can specify the level of protection to be applied to its communications with the server. The protection level determines how much of client/server messages are encrypted. Generally, the more restrictive the protection level, the greater the affect on performance. Different levels are provided so that applications can control the protection versus performance trade-offs.

The protection level is entirely a client responsibility. When a server registers its supported authentication service with the RPC runtime, it does not specify any protection information for that service. However, the server can include the protection level used for a particular operation when deciding if the caller is authorized to perform the operation.

Authenticated RPC supports the following protection levels:

| | |
|---|---|
| **rpc_c_protect_level_default** | Uses the default protection level for the specified authentication service. |
| **rpc_c_protect_level_none** | There is no protection level. |
| **rpc_c_protect_level_connect** | Performs protection only when the client establishes a relationship with the server. This level performs an encrypted communication signal exchange when the client first communicates with the server. Encryption or decryption is not performed on the data sent between the client and server, although the successful communication signal exchange indicates that the client is active on the network. |
| **rpc_c_protect_level_call** | Performs protection only at the beginning of each RPC when the server receives the request. This level attaches a verifier to each client call and server response.<br><br>This level does not apply to RPCs made over a connection-based protocol sequence, that is, **ncacn_ip_tcp**. If this level is specified and the binding handle uses a connection-based protocol sequence, the routine uses the **rpc_c_protect_level_pkt** level instead. |
| **rpc_c_protect_level_pkt** | Ensures that all data received is from the expected client. This level attaches a verifier to each message. |
| **rpc_c_protect_level_pkt_integ** | Ensures and verifies that none of the data transferred between client and server has been changed. This level computes a cryptographic checksum of each message to verify that none of the |

| | |
|---|---|
| | data transferred between the client and server has been changed in transit.<br><br>This is the highest protection level that is guaranteed to be present in the RPC runtime. |
| **rpc_c_protect_level_cdmf_priv** | Performs protection as specified by all of the previous levels and also encrypts each remote procedure call argument value. This level encrypts all user data in each call and provides a lower level of packet privacy than **rpc_c_protect_level_pkt_privacy**. This is the second highest protection level, but it is available only if one of the User Data Privacy optional features (DES and CDMF, or CDMF only) was installed. |
| **rpc_c_protect_level_pkt_privacy** | Performs protection as specified by all of the previous levels and also encrypts each RPC argument value. This level encrypts all user data in each call.<br><br>This is the highest protection level, but it may not be available in the RPC runtime. |

If a client wants to use the default protection level but does not know what this level is, it can use the **rpc_mgmt_inq_dflt_protect_level()** routine to determine the default level.

## Authorization

Authorization is the process of checking a client's permissions to an object that is controlled by the server. Access checking is entirely a server responsibility, and involves matching the client's privilege attributes against the permissions associated with the object. A client's privilege attributes consist of the principal ID and group memberships contained in the client's network login context.

Authenticated RPC supports the following options for making client authorization information available to servers for access checking:

| | |
|---|---|
| **rpc_c_authz_none** | No authorization information is provided to the server, usually because the server does not perform access checking. |
| **rpc_c_authz_name** | Only the client principal name is provided to the server. The server can then perform authorization based on the provided name. This form of authorization is sometimes referred to as *name-based* authorization. |
| **rpc_c_authz_dce** | The client's DCE Extended Privilege Attribute Certificate (EPAC) is provided to the server with each RPC made with binding parameter. The server performs authorization using the client EPAC. Generally, access is checked against DCE ACLs. |

When a client establishes authenticated RPC, it must indicate the authorization option that it wants to use.

It is the server's responsibility to carry out the type of authorization appropriate for the objects that it controls. When the server calls **rpc_binding_inq_auth_client()** to return information about an authenticated client, it gets back either the client's principal name or a pointer to the data structure that contains the client's EPAC. The value that is returned depends on the type of authorization the client specified on its call to establish authenticated RPC with that server.

Each server is responsible for carrying out its own access checking by means of ACL managers. When a server receives a client request for an object, the server runs the ACL manager appropriate for that type of object and passes to the manager, the client's authorization data. The manager compares the client authorization data to the permissions associated with the object and either refuses or permits the requested operation. For certified (EPAC-based) authorization, servers must carry out access checking using the ACL facility provided by the DCE Security Service. ACL managers are described in more detail in Part 5, “Using the DCE Security APIs” on page 395.

**Name-Based Authorization:** Name-based authorization (**rpc_c_authz_name**) provides a server with the client’s principal name. The server call to **rpc_binding_inq_auth_client()** retrieves the name from the binding handle associated with the client and returns it as a character string.

Do not use names for authorization. To perform access checking using client principal names, you must store the names in the access lists associated with the protected objects. Each time a name is changed, the change has to be spread through all the access lists in which the name is defined.

**DCE Authorization:** DCE authorization (**rpc_c_authz_dce**) provides a server with the client’s EPAC.

EPACs offer a trusted mechanism for conveying client authorization data to authenticated servers. The DCE Security Service generates a client EPAC in a tamper-proof manner. When a server receives a client EPAC, it knows that the EPAC has been certified by DCE Security.

EPACs are designed to be used with the DCE ACL facility. The ACL facility provides an editor and a set of API routines that support the implementation of access control lists and the managers to control them.

For a detailed description of EPACs and their use with DCE ACL facility, refer to Part 5, “Using the DCE Security APIs” on page 395.

## Authenticated RPC Routines

Authenticated RPC is carried out using a set of related RPC routines. Some of the routines are for use by clients, some are for use by servers and their managers, and some are for use by both clients and servers. The authenticated RPC routines are as follows:

| | |
|---|---|
| **rpc_binding_set_auth_info()** | A client calls this routine to establish an authentication service, protection level, and authorization service for a server binding handle. The client identifies the server by supplying the server’s principal name. The RPC runtime, in conjunction with the DCE Security Service, applies the authentication service and protection level to all subsequent remote procedure calls made using the binding handle. |
| **rpc_binding_inq_auth_info()** | A client calls this routine to return the authentication service, protection level, and authorization service that are in effect for a specified server binding handle. This routine also returns the principal name of the server associated with the binding handle. |
| **rpc_mgmt_inq_dflt_protect_level()** | A client or a server calls this routine to learn the default protection level that is in force for a given authentication service. |
| **rpc_mgmt_inq_server_princ_name()** | A client, a server, or a server manager can call this routine to return the principal name that a server has registered with the RPC runtime with the **rpc_server_register_auth_info()** routine. A client can identify the desired server by supplying a server binding handle and the authentication service associated with the registered principal name. |

|  |  |
|---|---|
| **rpc_server_register_auth_info()** | A server calls this routine to register an authentication service that it wants to support and the server principal name to be associated with the registered service. The server can also supply the address of a key retrieval routine to be called by the DCE Security Service as part of the client authentication process. The routine is a user-supplied function whose purpose is to provide the server's key to the DCE Security runtime.<br><br>Note that the server registers only an authentication service. It does not establish a protection level or an authorization service. These are the responsibilities of the client. |
| **rpc_binding_inq_auth_caller()** | A server calls this routine to return the authentication service, protection level, and authorization service that is associated with the binding handle of an authenticated client. This call also returns the server principal name specified by the client on its call to **rpc_binding_set_auth_info()**. |
| **rpc_binding_inq_auth_client()** | A server calls this routine to return the authentication service, protection level, and authorization service that is associated with the binding handle of an authenticated client. This call also returns the server principal name specified by the client on its call to **rpc_binding_set_auth_info()**.<br><br>**Note:** This call is provided only for compatibility purposes with DCE 1.1 applications. DCE release 1.1 and later applications should use **rpc_binding_inq_auth_caller()**. |
| **rpc_mgmt_set_authorization_fn()** | A server calls this routine to establish a user-supplied authorization function to validate remote client calls to the server's management routines. For example, the user function can call **rpc_binding_inq_auth_client()** or **rpc_binding_inq_auth_caller()** to return authentication and authorization information about the calling client. The RPC runtime calls the user-supplied function whenever it receives a client request to run one of the following server management routines:<br>• **rpc_mgmt_inq_if_ids()**<br>• **rpc_mgmt_inq_server_princ_name()**<br>• **rpc_mgmt_inq_stats()**<br>• **rpc_mgmt_is_server_listening()**<br>• **rpc_mgmt_stop_server_listening()**. |

When an unauthenticated client calls a server that has specified authentication, the RPC runtime will not perform any authentication, and the call will reach the application manager code. It is up to the manager to decide how to deal with the unauthenticated call.

Typically, servers and clients establish authentication as follows:

- The server specifies an authentication service for a principal identity under which it runs by calling **rpc_server_register_auth_info()**. The authentication service is specified by the *authn_svc* parameter of this call. Servers may specify either DCE secret key authentication (by supplying either **rpc_c_authn_dce_secret** or **rpc_c_authn_default**) or no authentication (by supplying **rpc_c_authn_none**). The specified authentication service will be used if it is also requested by the client.

- The client sets authentication for a binding handle by calling **rpc_binding_set_auth_info()**. The choices are also either DCE secret key or no authentication. Client calls made on the binding handle attempt to use the specified authentication service.
- The server manager code calls **rpc_binding_inq_auth_client()** to extract any authentication information from the client binding for the call.

## Directory Services and RPC: Using the Namespace

This section discusses how the DCE RPC Name Service Interface (NSI) configures directory service entries and how RPC applications can use those entries. The following topics are included:

- Directory service entries defined by NSI

  Describes the kinds of directory service entries NSI defines
- Searching the Namespace

  Describes how the namespace is searched when a client requests binding information
- Strategies for using directory service entries

  Outlines strategies for using each kind of entry
- The Service Model

  Describes the service model for defining RPC servers and introduces NSI usage models intended to guide application developers in assessing how to best use NSI for a given application
- The Resource Model

  Describes the resource model for defining RPC servers

### NSI Directory Service Entries

To store information about RPC servers, interfaces, and objects, NSI defines the following directory service entries in the namespace: server entries, groups, and profiles. These directory service entries are CDS objects.

- A server entry is a directory service entry that stores binding information and object UUIDs for an RPC server.
- A group is a directory service entry that corresponds to one or more RPC servers that all offer certain RPC interfaces or RPC object types (or both).
- A profile is a directory service entry that defines search paths in a namespace for a server that offers a particular RPC interface and object.

The use of server entries, groups, and profiles determines how clients view servers. A server describes itself to its clients by exporting binding information associated with interfaces and objects to one or more server entries. A group corresponds to servers that offer a given interface, service, or object. Profiles enable clients to access alternative directory service entries when searching for an interface or object. Used together, groups and profiles offer sophisticated ways for RPC applications to maintain and use directory service data.

**NSI Attributes:** Usually, the distinct server entries, groups, and profiles concepts are adequate for using the NSI. However, you can combine server entries, groups, and profiles into a single directory service entry because of the way the NSI stores RPC information. To store information about RPC applications in a directory service entry, the RPC directory service interface defines several RPC-specific directory service attributes, or NSI attributes. NSI attributes contain information about RPC applications in a directory service entry. The NSI attributes are as follows:

- **NSI Binding attribute**

  The binding attribute stores binding information and interface identifiers (interface UUID and version numbers) exported to the server entry. This attribute identifies a directory service entry as a server entry.

- **NSI Object attribute**

  The object attribute stores a list of one or more object UUIDs. Whenever a server exports any object UUIDs to a server entry, the server entry contains an object attribute as well as a binding attribute. When a client imports a binding from that entry, the import operation returns an object UUID from the list stored in the object attribute.

- **NSI group attribute**

  The group attribute stores the entry names of the members of a single group. This attribute identifies a directory service entry as an RPC group.

- **NSI profile attribute**

  The profile attribute stores a set of profile elements. This attribute identifies a directory service entry as an RPC profile.

Figure 23 represents the correspondence between NSI attributes and the different directory service entries: server entries, groups, and profiles.

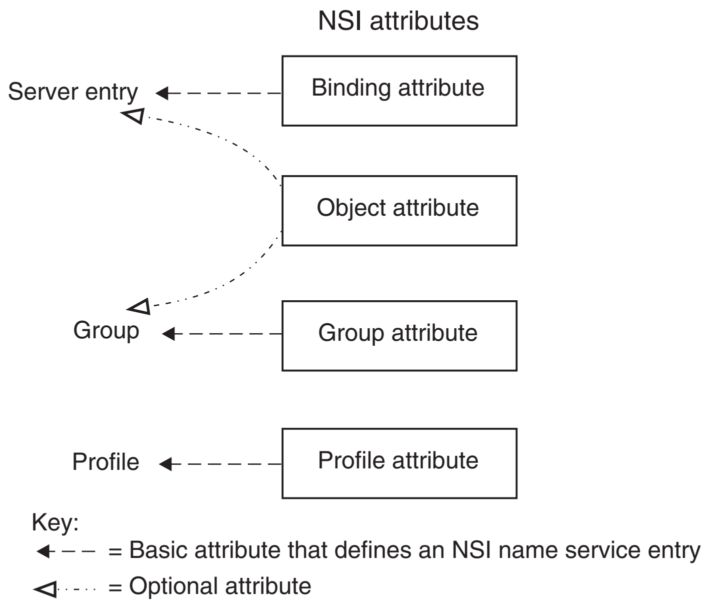


*Figure 23. NSI Attributes*

Any directory service entry can contain any combination of the four NSI attributes. However, to facilitate administrating directory service entries, avoid creating binding, group, and profile attributes in the same entry. Instead, use distinct directory service entries for server entries, groups, and profiles. The object attribute, in contrast, is designed as an adjunct to another NSI attribute, especially the binding attribute.

When using the resource model or when distinguishing server instances, a server entry contains an object attribute as well as a binding attribute. On finding a server entry whose binding attribute contains compatible binding information, an NSI search operation also looks in the entry for an object attribute. For groups whose membership is selected according to a shared object or set of objects, it may be useful to export those objects to the group. In this case, the directory service entry of the group contains both group and object attributes. For reading the object UUIDs in the NSI object attribute in any directory service entry, the NSI provides a set of object inquiry operations called using the **rpc_ns_entry_object_inq_**{**begin**,**next**,**done**}**()** routines.

Using separate entries facilitates administration of the namespace, for example, by enabling entry names to specifically describe their contents. Keeping server entries, profiles, and groups separate allows clear references to each of them.

**Note:** In addition to any NSI attributes, a directory service entry contains other kinds of directory service attributes. Every entry in a namespace contains standard attributes created by the directory service. NSI operations rely on some standard attributes to identify and use an entry. Any directory service entry can also contain additional attributes specified by non-RPC applications; these are ignored by NSI operations.

**Structure of Entry Names:** Each entry in a namespace is identified by a unique *global name* comprising a cell name and a cell-relative name.

A *cell* is a group of users, systems, and resources that share common DCE services. A cell configuration includes at least one Cell Directory Server, one Security Server, and one Distributed Time Server. A cell's size can range from one system to thousands of systems. A host is assigned to its cell by a DCE configuration file. For information on cells, see the *z/OS DCE Administration Guide*.

The following is an example of a global name:

```
/.../C=US/O=uw/OU=MadCity/LandS/anthro/Stats_host_2
```

The parts of a global name are as follows:

- Cell name (using X.500 name syntax):

  ```
  /.../C=US/O=uw/OU=MadCity
  ```

  The symbol **/...** begins a *cell name*. The letters before the **=** (equal signs) are abbreviations for Country (C), Organization (O), and Organization Unit (OU). For entries in the local cell, the cell name can be represented by a **/.:** prefix, in place of the actual cell name, for example:

  ```
  /.:/LandS/anthro/Stats_host_2
  ```

  The **/** (slash) to the right of the cell name represents the root of the cell directory (the *cell root*).

  For NSI operations on entries in the local cell you can omit the cell name.

- Cell-relative name (using DCE name syntax):

  Each directory service entry requires a cell-relative name, which contains a directory path name and a leaf name.

  – A *directory path name* follows the cell name and indicates the hierarchical relationship of the entry to the cell root.

    The directory path name contains the names of any subdirectories in the path; each subdirectory name begins with a **/** (slash), as follows:

    /*sub-dir-a-name*/*sub-dir-b-name*/*sub-dir-c-name*

Directory path names are created by directory service administrators. If an appropriate directory path name does not exist, ask your directory service administrator to extend an existing path name or create a new one. In a directory path name, the name of a subdirectory should reflect its relationship to its *parent directory* (the directory that contains the subdirectory).

– A *leaf name* identifies the specific entry.

  The leaf name constitutes the right-hand part of a global name beginning with the rightmost **/** (slash).

For example, `/.:/LandS/anthro/Cal_host_4`, where `/.:/` represents the cell name, `/LandS/anthro` is the directory path name, and `/Cal_host_4` is the leaf name. If the directory service entry is located at the cell root, the leaf name directly follows the cell name, for example, `/.:/cell-profile`.

**Note:** When the NSI is used with CDS, the cell-relative name is a CDS name.

Figure 24 shows the parts of a global name.

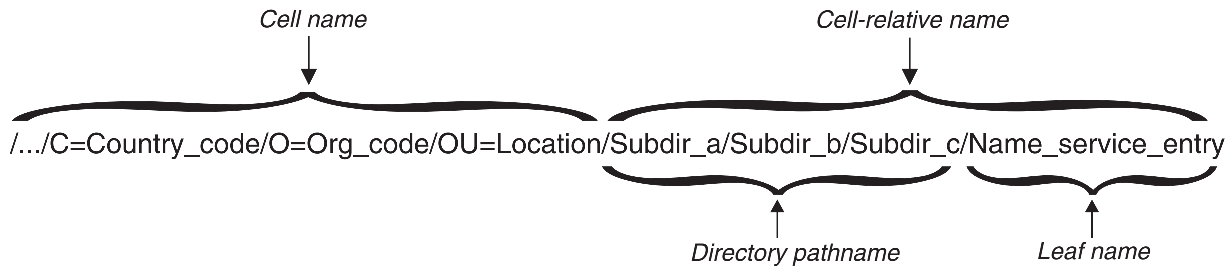

Cell name
Cell-relative name
/.../C=Country_code/O=Org_code/OU=Location/Subdir_a/Subdir_b/Subdir_c/Name_service_entry
Directory pathname
Leaf name

*Figure 24. Parts of a Global Name*

**Server Entries:** The NSI enables any RPC server or the DCE administrator with the necessary directory service permissions to create and maintain server entries in a name service database. A server can use as many server entries as it needs to advertise combinations of its RPC interfaces and objects.

Each server entry must correspond to a single server (or a group of interchangeable server instances) on a given system. *Interchangeable server instances* are instances of the same server running on the same system that offer the same RPC objects (if any). Only interchangeable server instances can share a server entry.

Each server entry must contain binding information. Every combination of protocol sequence and network addressing information represents a potential binding. The network addressing information can contain a network address but lacks an endpoint, making the address partially bound. A server entry can also contain a list of object UUIDs exported by the server. Each of the object UUIDs corresponds to an object offered by the server. In a given server entry, the binding information is stored for each interface UUID, and each interface identifier is associated with every object UUID.

Figure 25 on page 100 represents a server entry. This server entry was created by two calls to the **rpc_ns_binding_export()** routine. The first call created the first column of the top half of the figure. The routines's *binding_vec* parameter has three elements, each of which is paired with the routine's *if_handle* parameter. The ellipsis points under the last box indicate that more elements in the routine's *binding_vec* argument would result in more interface UUID/binding information pairs in the first column.

The second call to the **rpc_ns_binding_export()** routine created the second column of the top half of the figure. The routine's *binding_vec* parameter has two elements, each of which is paired with the routine's *if_handle* parameter. For example, the first element could have contained binding information with the **ncacn_ip_tcp** protocol sequence, and the second element could have contained binding information with the **ncadg_ip_udp** protocol sequence. As in the first column, more elements in the routine's *binding_vec* parameter would result in more interface UUID/binding information pairs.

Subsequent calls to the **rpc_ns_binding_export()** routine would create more columns; the horizontal ellipsis points indicate this expansion.

The **rpc_ns_binding_export()** routine optionally takes a vector of object UUIDs. The four object UUIDs in the bottom half of the figure came from the two calls to the routine, or from another call to the routine with no interface UUID/version and with no binding information, but with object UUIDs. The object UUIDs are associated with no particular binding. Instead, they are associated with all the bindings. Subsequent calls to the routine could create more object UUIDs; the three vertical dots indicate this expansion.

**Note:** To distinguish among RPC objects when using the CDS ACL editor, export the RPC objects to separate directory service entries.

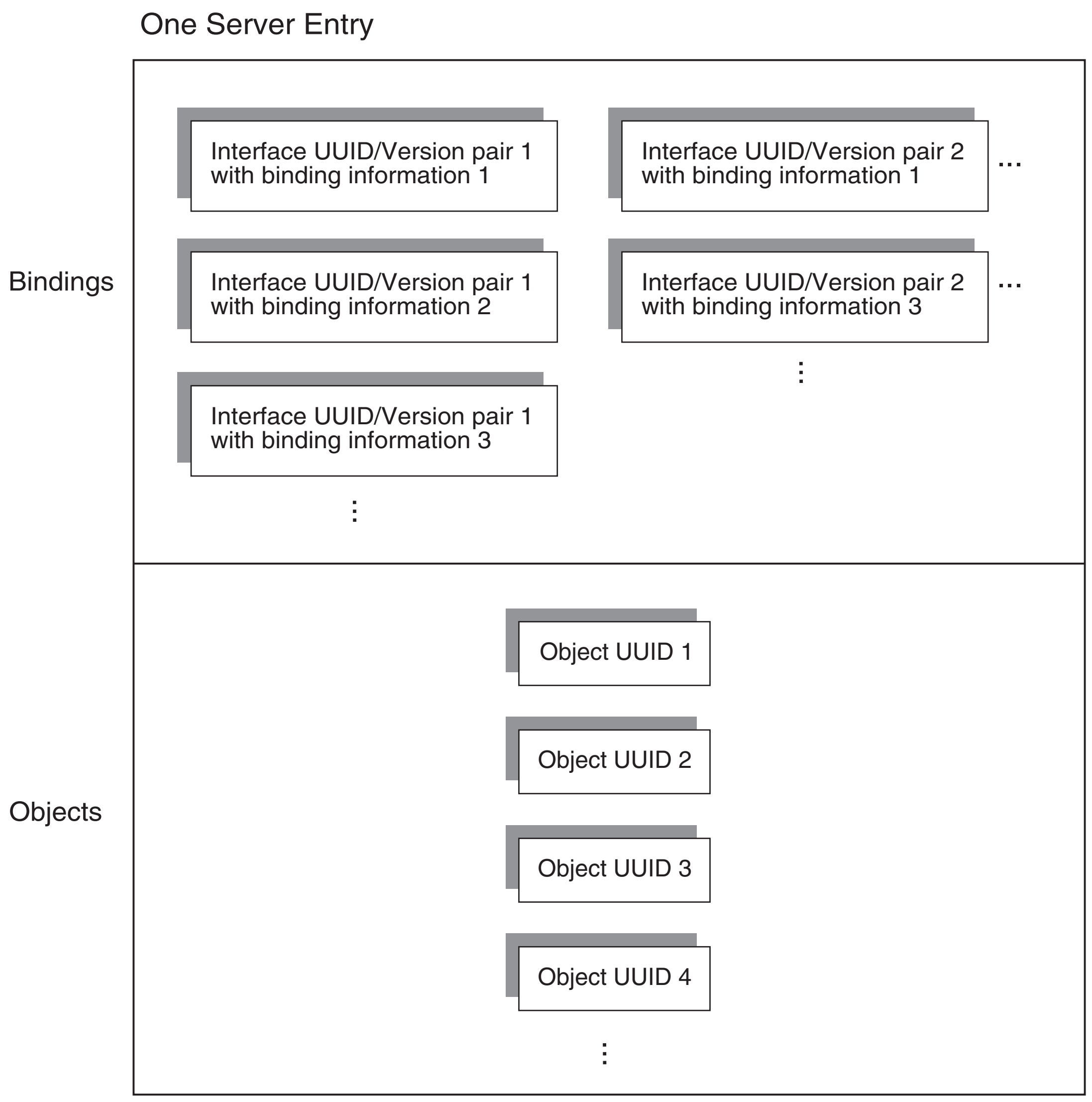

One Server Entry
Bindings
Interface UUID/Version pair 1 with binding information 1
Interface UUID/Version pair 2 with binding information 1
Interface UUID/Version pair 1 with binding information 2
Interface UUID/Version pair 2 with binding information 3
Interface UUID/Version pair 1 with binding information 3
Objects
Object UUID 1
Object UUID 2
Object UUID 3
Object UUID 4

*Figure 25. Possible Information in a Server Entry*

**Groups:** Administrators or users of RPC applications can organize searches of a namespace for binding information by having clients use an RPC group as the starting point for NSI search operations. A group provides NSI *search operations* (**rpc_ns_binding_import_next** or **rpc_ns_binding_lookup_next** operations) with access to the server entries of different servers that offer a common RPC interface or object. A group contains names of one or more server entries, other groups, or both. Because a group can contain group names, groups can be nested. Each server entry or group named in a group is a member of the group. A group's members must offer one or more RPC interfaces, the type of RPC object, or both in common.

Figure 26 shows an example of the kinds of members a group can contain and how they correspond to database entries.

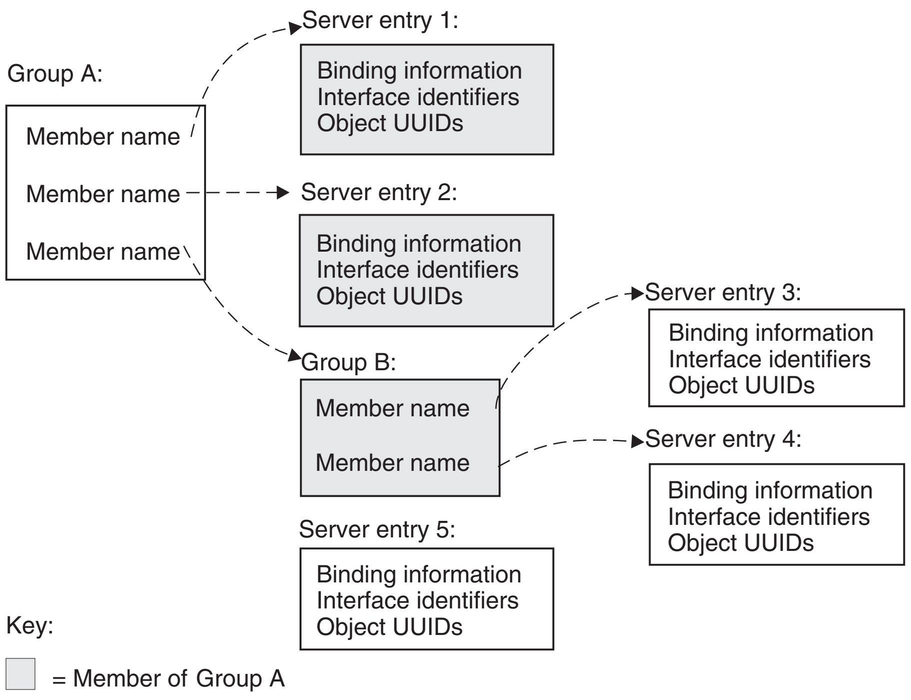


*Figure 26. Possible Mappings of a Group*

The members of Group A are Server Entry 1, Server Entry 2, and Group B. The members of the nested group, Group B, are Server Entry 3 and Server Entry 4. An additional server entry that advertises the common interface or object, Server Entry 5, is omitted from either group.

**Profiles:** Administrators or users of RPC applications can use profiles to organize searches of a namespace to find binding information. Clients can use an RPC profile as the starting point for NSI search operations. A *profile* is an entry in a namespace that contains a collection of profile elements. A *profile element* is a database record that associates a single RPC interface with a server entry, group, or profile. Each profile element contains the following information:

- Interface identifier

  This field is the key of the profile. The *interface identifier* consists of the interface UUID and the interface version numbers.

- Member name

  The entry name of one of the following kinds of directory service entries:

  – A server entry for a server offering the requested RPC interface

  - A group corresponding to the requested RPC interface
  - A profile.

- Priority value

  The *priority value* (0 is the highest priority; 7 the lowest) is designated by the creator of a profile element to help determine the order for using the element. NSI search operations will select among like-priority elements at random.

  Note that when you create an NSI profile, you should use priorities 1 through 7 to manage profile elements. Priority 0 is a reserved priority that should only be used by the default profile element. If priority 0 is used by profile elements other than the default profile element, then DCE will choose a random binding from among the two. DCE does not always distinguish between priority 0 default elements and priority 0 nondefault elements. Therefore, if you create an NSI profile it is better to use priority 1 to denote the highest priority element in your profile instead of using priority 0. The other lower priority elements in your profile can be given priorities 2 through 7 leaving priority 0 reserved for the default element. This convention along with the fact that only one default element is allowed per profile will prevent the high priority given to default elements from being abused.

- Annotation string

  The *annotation string* enables you to identify the purpose of the profile element. The annotation can be any textual information, for example, an interface name associated with the interface identifier or a description of a service or resource associated with a group.

  Unlike the interface identifier field, the annotation string is not a search key.

Optionally, a profile can contain one default profile element. A *default profile element* is the element that an NSI search operation uses when a search using the other elements of a profile finds no compatible binding information, for example, when the current profile lacks any element corresponding to the requested interface. A default profile element contains the nil interface identifier, a priority of 0, the entry name of a default profile, and an optional annotation.

A *default profile* is a backup profile, referred to by a default profile element in another profile. A profile designated as a default profile should be a comprehensive profile maintained by an administrator for a major set of users, such as the members of an organization or the owners of computer accounts on a LAN.

A default profile must not create circular dependencies between profiles. For example, when a public profile refers to an application's profile, the application's profile must not specify that public profile as a default profile.

Figure 27 on page 103 shows an example of the kinds of elements a profile can contain and how they correspond to database entries.

.

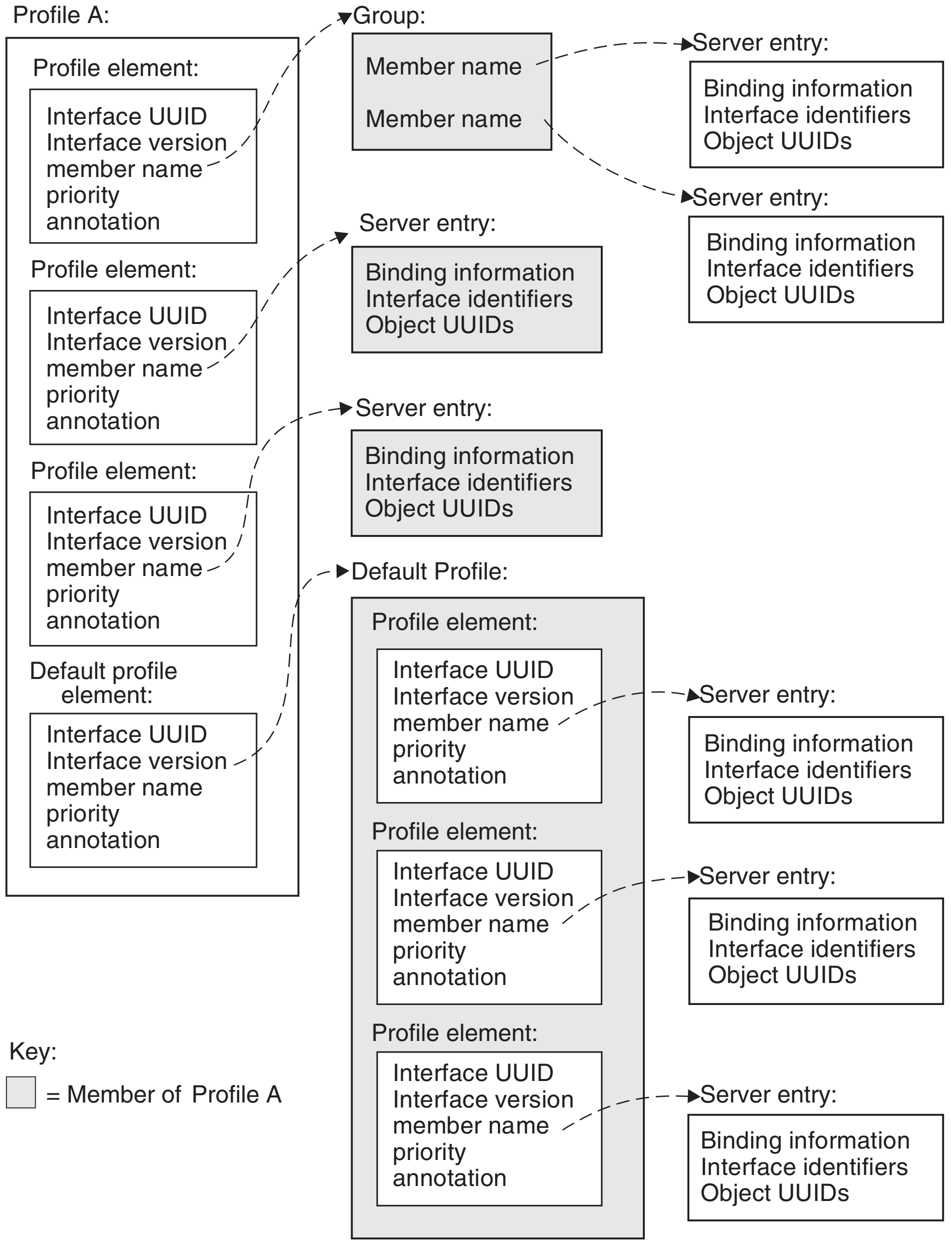


*Figure 27. Possible Mappings of a Profile*

NSI search operations use a profile to construct an *NSI search path*. When an NSI search operation reads a profile, the operation dynamically constructs its NSI search path from the set of elements that correspond to a common RPC interface.

A profile element is used only once per NSI search path. The construction of NSI search paths depends partly on the priority rankings of the elements. A search operation uses higher-priority elements before lower-priority elements. Elements of equal priority are used in random order, permitting some variation in the NSI search paths between searches for a given interface. If nondefault profile elements do not satisfy a search, the search path extends to the default profile element, if any.

Profiles meet the needs of particular individuals, systems, LANs, sites, organizations, and so forth, with minimal configuration management. The administrator of a profile can set up NSI search paths that reflect the preferences of the profile's user or users. The profile administrator can set up profile elements that refer (directly or indirectly) to only a subset of the server entries that offer a given RPC interface. The administrator can also assign different search priorities to the elements for an interface.

**Guidelines for Constructing Names of Directory Service Entries:** A global name includes both a cell name and a cell-relative name composed of a directory path name and a leaf name. The cell name is assigned to a cell root at its creation. When you specify only a cell-relative name to an NSI operation, the NSI automatically expands the name into a global name by inserting the local cell name. When returning the name of a directory service entry, a group member, or a member in a profile element, NSI operations return global names.

The directory path name and leaf name uniquely identify a directory service entry. The leaf name should somehow describe the entry, for example, by identifying its owner or its contents. The remainder of this section contains guidelines for choosing leaf names.

**Note:** Directory path names and leaf names are case sensitive.

Use the guidelines in the following three subsections for constructing names.

***Naming a Server Entry:*** For a server entry that advertises an RPC interface or service offered by a server, the leaf name must distinguish the entry from the equivalent entries of other servers. When a single server instance runs on a host, you can ensure a unique name by combining the name of the service, interface (from the interface definition), or the system name for the server's host system.

For example, consider two servers, one offering a calendar service on host JULES and one on host VERNE.

The server on JULES uses the following leaf name:

```
calendar_JULES
```

The server on VERNE uses the following leaf name:

```
calendar_VERNE
```

For servers that perform tasks on or for a specific system, an alternative approach is to create server entries in a system-specific host directory within the namespace. Each host directory takes the name of the host to which it corresponds. Because the directory name identifies the system, the leaf name of the server entry name does not need to include the host name, for example:

```
/.:/LandS/host_1/Process_control
```

To construct names for the server entries used by distinctive server instances on a single host, you can construct unique server entry names by combining the following information: the name of the server's service, interface, or object; the system name of the server's host system; and a reusable instance identifier such as an integer.

For example, the following leaf names distinguish two instances of a calendar service on the JULES system:

```
calendar_JULES_01

calendar_JULES_02
```

Avoid automatically generating entry names for the server entries of server instances, for example, by using unique data such as a timestamp (`calendar_verne_15OCT91_21:25:32`) or a process identifier (`calendar_jules_208004D6`). When a server incorporates such unique data into its server entry names, each server instance creates a separate server entry, causing many server entries. When a server instance stops running, it leaves an obsolete server entry that is not reused. The creation of a new entry whenever a server instance starts may impair performance.

A server can use multiple server entries to advertise different combinations of interfaces and objects. For example, a server can create a separate server entry for a specific object and the associated interfaces. The name of such a server entry should correspond to a well-known name for the object. For example, consider a server that offers a horticulture bulletin board known to users as `horticulture_bb`. The server exports the `horticulture_bb` object, binding information, and the associated bulletin-board interface to a server entry whose leaf name identifies the object, as follows:

```
horticulture_bb
```

***Naming a Group:*** The leaf name of a group should indicate the interface, service, or object that determines membership in the group. For example, for a group whose members are selected because they advertise an interface named Statistics, the following is an effective leaf name:

```
Statistics
```

For a group whose members advertise laser printer print queues as objects, the following is an effective leaf name:

```
laser-printer
```

***Naming a Profile:*** The leaf name of a profile should indicate the profile users. For example, for a profile that serves the members of an accounting department, the following is an effective leaf name:

```
accounting_profile
```

***NSI Begin, Next, and Done Operations:*** NSI accesses a variety of search and inquire operations that read NSI attributes in directory service entries. An NSI attribute is an RPC-defined attribute of a directory service entry used by the DCE RPC directory service interface; an NSI attribute stores one of the following:

- Binding information
- Object UUIDs
- A group
- Members
- Profile elements.

Reading information from any attribute involves an equivalent set of search or inquire operations; that are an integral set of **begin**, **next**, and **done** operations. An RPC application uses these operations as follows:

1. The application creates a *name service handle* (a reference to the context of the ensuing series of **next** operations) by calling an NSI **begin** operation.
2. The application calls the NSI **next** operation that corresponds to the **begin** operation one or more times. Each **next** operation returns another value or list of values from the target RPC directory service attribute. For example, an **import_next** operation returns binding information from a binding attribute and an object from an object attribute.

   Each call to an NSI **next** operation requires the directory service handle created in the associated NSI **begin** operation. The directory service handle maintains state information for reading values from directory service attributes; this is analogous to the function of a file pointer in the C language.
3. The application deletes the directory service handle by calling the corresponding NSI **done** operation.

**Note:** Search and inquire operations are also accessible interactively from within the RPC control program.

The NSI **next** operations used by RPC applications are listed as follows:

| **Search Operation** | **Attributes Traversed** |
|---|---|
| **rpc_ns_binding_import_next()** | Searches for binding and object attributes of a *compatible server* and reads any NSI attribute in a search path. A compatible server is a server that supports the same interface or object, and the same transport protocol as the client.<br><br>Returns one binding handle that refers to a potential binding for a compatible server. |
| **rpc_ns_binding_lookup_next()** | Searches for binding and object attributes of a compatible server; reads any NSI attribute in a search path.<br><br>Returns a vector of binding handles, each of which refers to a potential binding for a compatible server.<br><br>Note that after calling the **rpc_ns_binding_lookup_next** operation, the client must select one binding handle from the list. To select a binding handle at random, the client can call the NSI binding select routine (**rpc_ns_binding_select()**). For an alternative selection algorithm, the client can define and call its own application-specific select algorithm. |

| **Inquire Operation** | **Attributes Traversed** |
|---|---|
| **rpc_ns_group_mbr_inq_next()** | Reads a group attribute and returns a compatible member name. |
| **rpc_ns_profile_elt_inq_next()** | Reads a profile attribute and returns the fields of a compatible profile element. |

**Selecting the Starting Entry:** When searching a namespace for an RPC interface and object, a client supplies the name of the directory service entry where the search begins. The entry can be a server entry, group, or profile. Generally, an NSI search starts with a group or profile. The group or profile defines a search path that ends at a server entry containing the requested interface identifier, object UUID, and compatible binding information.

A user may know in advance what server instance to use. In this case, starting with a server entry for the server instance is appropriate.

**Environment Variables:** DCE RPC provides predefined environment variables that a client can use for NSI operations. An *environment variable* is a variable that stores information, such as a name, about a particular environment. The NSI interface uses two environment variables, **RPC_DEFAULT_ENTRY** and **RPC_DEFAULT_ENTRY_SYNTAX**. They describe the NSI starting entry and its syntax.

When a client searches for binding information, the search starts with a specific entry name. Optionally, a client can specify this entry name as the value of the **RPC_DEFAULT_ENTRY** variable. A client can also specify the name syntax of the starting entry as the value of the **RPC_DEFAULT_ENTRY_SYNTAX** variable; the default name syntax is **dce**.

**Note:** The **dce** name syntax is the only syntax supported by the DCE Cell Directory Service. Because the NSI is independent of any specific directory service, it may support one or more alternative directory services that use different name syntaxes.

**Searching a Namespace for Binding Information:** Searching the namespace for binding information requires that a client specify a starting point for the search. A client can start with a specific server entry. However, this is a limiting approach because the client is restricted to using one server. To avoid this restriction, a client can start searching with a group or a profile instead of with a server entry. Searches that start with a profile or a group should encounter the server entry of a compatible server. If such an entry is not encountered, a search operation returns the **rpc_s_no_more_bindings** status code to the client. When calling the **rpc_ns_binding_import_next()** or **rpc_ns_binding_lookup_next()** routine, a client must track whether the routine returns this status code.

**The Import and Lookup Search Algorithm:** The NSI search operations (the **rpc_ns_binding_import_next** and **rpc_ns_binding_lookup_next** operations) traverse one or more entries in the namespace when searching for compatible binding information. In each directory service entry, these operations ignore non-RPC attributes and process the NSI attributes in the following order:

1. Binding attribute (and object attribute, if present)
2. Group attribute
3. Profile attribute.

If an NSI search path includes a group attribute, the path can encompass every entry named as a group member. If a search path includes a profile attribute, it can encompass every entry named as the member of a profile element that contains the target interface identifier. A search finishes only when it finds a server entry containing compatible binding information and the non-nil object UUID, if requested. Search operations take the following steps when traversing a directory service entry:

**Step 1: Binding attribute**

In each entry, the search operation starts by searching for a compatible interface identifier in the binding attribute, if present.

The absence of a binding attribute or of any compatible interface identifier causes the search operation to go directly to step 2.

The presence of any compatible interface identifier indicates that compatible potential bindings may exist in the binding attribute. At this point, object UUIDs may affect the search, as follows:

- If the client specified the nil object UUID, object UUIDs do not affect the success or failure of the search. The search returns compatible binding information for one or more potential bindings.
- If the client specified a non-nil object UUID, the search reads the object attribute, if present, to look for the requested object UUID. This search for an object UUID has one of the following outcomes:
  - On finding the specified object UUID, the search returns the object UUID along with compatible binding information for one or more potential bindings.
  - If a requested object UUID is absent, the search continues to step 2.

**Note:** If a search involves a series of **import_next** or **lookup_next** operations, a subsequent next operation resumes the search at the point in the search path where the preceding operation left off.

**Step 2: Group attribute**

If the binding attribute does not lead to compatible binding information, or if a series of import next or lookup next operations uses up the compatible binding information, the search continues by reading the group attribute, if present. If the directory service entry lacks a group attribute, the search goes directly to step 3.

The search operation selects a member of the group at random, goes to the entry of that member, and resumes the search at step 1. Unless a group member leads the search to compatible binding information, the search looks at all the members of the group, one by one in random order, until none remain.

**Step 3: Profile attribute**

If the binding and group attributes do not lead to compatible binding information, the search continues by reading the profile attribute, if present. If the directory service entry lacks a profile attribute, the search fails.

The search operation identifies all the profile elements containing the requested interface identifier and searches them in order of their priority beginning with the 0 (zero) priority elements. Profile elements of a given priority are searched in random order. For the selected profile element, the search reads the member name and goes to the corresponding directory service entry. There, the search resumes at step 1. Unless a profile element leads the search to compatible binding information, the search eventually looks at all the profile elements with the requested interface identifier, one by one, until none remain.

If the starting entry does not contain NSI attributes, or if none of the steps satisfies the search, the search operation returns an **rpc_s_no_more_bindings** status code to the client.

**Note:** The inquire next (**inq_next**) operations for objects, groups, or profiles look at only the entry specified in its corresponding inquire begin (**inq_begin**) operation. The search ignores nested groups or nested profiles.

The flow chart in Figure 28 on page 109 illustrates the three steps of the **rpc_ns_binding_import_next** and **rpc_ns_binding_lookup_next** search operations.

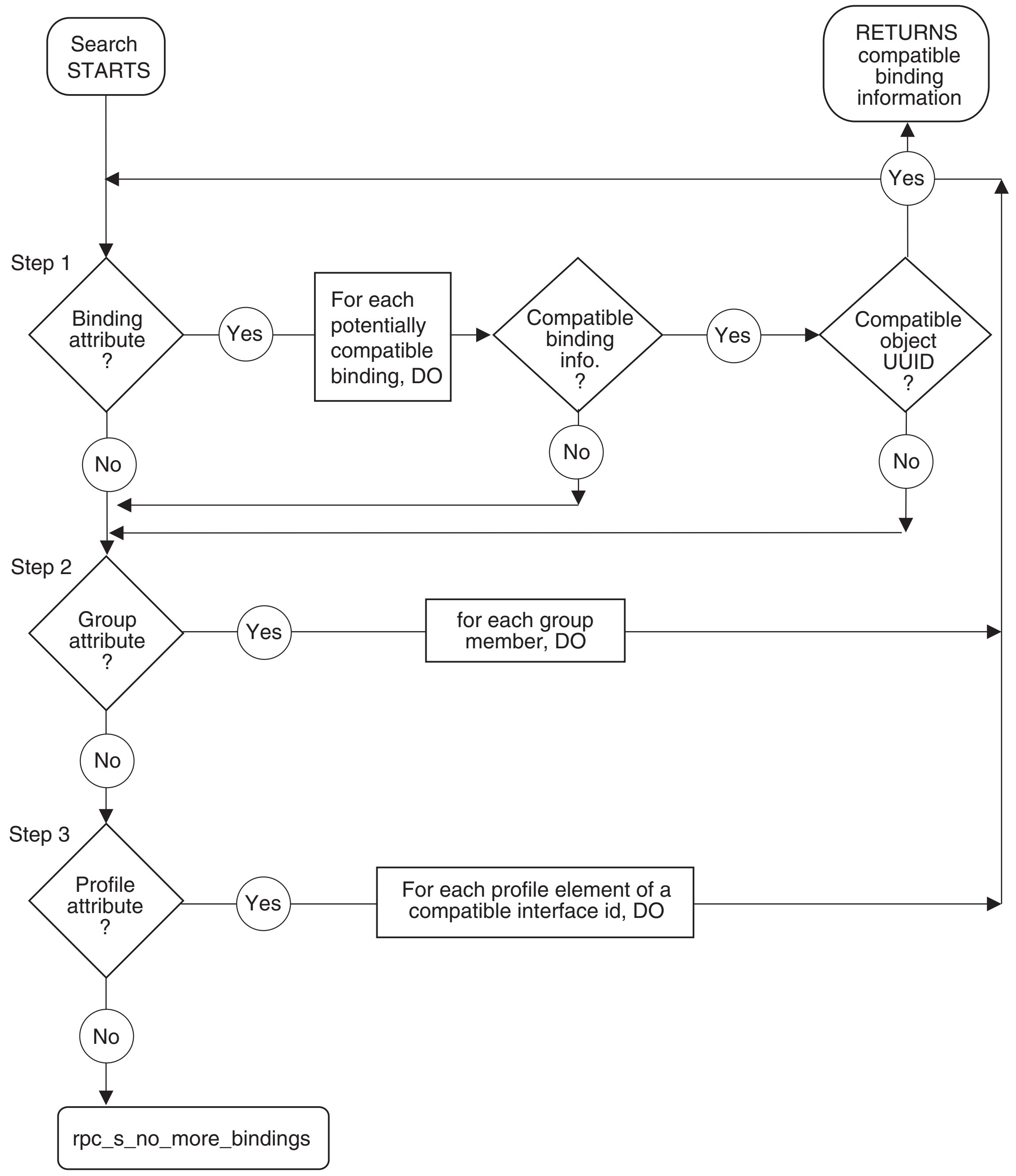

Search STARTS
RETURNS compatible binding information
Yes
Step 1
Binding attribute ?
Yes
For each potentially compatible binding, DO
Compatible binding info. ?
Yes
Compatible object UUID ?
No
No
No
Step 2
Group attribute ?
Yes
for each group member, DO
No
Step 3
Profile attribute ?
Yes
For each profile element of a compatible interface id, DO
No
rpc_s_no_more_bindings

*Figure 28. Import and Lookup Search Algorithm within a Single Entry*

**Note:** The inquire next (**inq_next**) operations for objects, groups, or profiles look at only the entry specified in its corresponding inquire begin (**inq_begin**) operation. The search ignores nested groups or nested profiles.

**Examples of Searching for Server Entries:** This subsection contains several examples of how the NSI **import** and **lookup** operations search for binding information associated with a given RPC interface and object in a namespace.

The following examples use the following conventions:

- To simplify the examples, each member name is represented by a leaf name preceded by the symbol that represents the local cell (`/.:`). For example, the full global name of the group for the `bulletin_board_interface` is:

  `/.../C=US/O=uw/OU=MadCity/LandS/bb_grp`

  The abridged name is `/.:/LandS/bb_grp`.

  **Note:** For a summary of global name syntax, see "Structure of Entry Names" on page 98 on naming directory service entries.

- Except for the nil interface UUID of the default profile, the examples avoid string representations of actual UUIDs. Instead, the examples represent a UUID as a value consisting of the name of the interface and the string *if-uuid*, or of the name of the object and the string *object-uuid*. For example:

  *calendar-if-uuid*,`1.0`

  *laser-printer-object-uuid*

- Profile elements in the examples are organized as follows (annotations are not displayed):

  *interface-identifier   member-name   priority*

  For example,

  `2fac8900-31f8-11ca-b331-08002b13d56d,1.0 /.:/LandS/C_host_7   0`

  which in the following examples is represented as:

  *calendar-if-uuid*,`1.0     /.:/LandS/C_host_7   0`

  **Note:** The priority is a value of 0 to 7, with 0 having the highest search priority and 7 having the lowest.

The first two examples begin with the personal profile of a user, Molly O'Brian, whose user name is `molly_o` and whose profile has the leaf name of `molly_o_profile`. To use this profile, Molly must specify its entry name to the client. Usually, a client either uses the predefined RPC environment variable **RPC_DEFAULT_ENTRY** or prompts for an entry name. For a client to use **RPC_DEFAULT_ENTRY**, the client or user must have already set the variable to a directory service entry.

The following example illustrates six profile elements from the individual user profile used in the first two examples. The six elements include five nondefault elements for some frequently used interfaces and a default profile element. Each profile element is displayed on three lines, but in an actual profile all the fields occupy a single record. The fields are the interface identifier (interface UUID and version numbers), member name, priority, and annotation.

```
/.:/LandS/anthro/molly_o_profile contents:

ec1eeb60-5943-11c9-a309-08002b102989,1.0
   /.../C=US/O=uw/OU=MadCity/LandS/Cal_host_7
   0  Calendar_interface_V1.0

ec1eeb60-5943-11c9-a309-08002b102989,2.0
   /.../C=US/O=uw/OU=MadCity/LandS/Cal_host_4
   1  Calendar_interface_V2.0

62251ddd-51ed-11ca-852c-08002b1bb4f6,2.0
   /.../C=US/O=uw/OU=MadCity/bb_grp
   2  Bulletin_board_interface_V2.0
```

```
62251ddd-51ed-11ca-852c-08002b1bb4f6,2.1
   /.../C=US/O=uw/OU=MadCity/bb_grp
   3  Bulletin_board_interface_V2.1

9e18d295-51ec-11ca-9cc0-08002b1bb4f5,1.0
   /.../C=US/O=uw/OU=MadCity/LandS/anthro/Zork_host_2
   0  Zork_interface_V1.0

00000000-0000-0000-0000-000000000000,0.0
   /.../C=US/O=uw/OU=MadCity/cell-profile
   0  Default_profile_element
```

**Example 1: Importing for an Interface with Multiple Versions**

**Target Interface:** Calendar V2.0

1. The search for binding information associated with Calendar V2.0 starts with the entry `molly_o_profile`:

   `/.../C=US/O=uw/OU=MadCity/LandS/anthro/molly_o_profile contents:`

   *calendar-if-uuid*,1.0   `/.:/LandS/C_host_7   0`

   *calendar-if-uuid*,2.0   `/.:/LandS/C_host_4   1`

   *bulletin_board-if-uuid*,2.0   `/.:/LandS/bb_grp   2`

   *bulletin_board-if-uuid*,2.1   `/.:/LandS/bb_grp   3`

   *Zork-if-uuid,*1.0   `/.:/Eng/Zork_host_2   0`

   `00000000-0000-0000-0000-000000000000,0.0   /.:/cell_ profile   0`

   The search operation examines only the two profile elements that refer to the Calendar interface:

   a. The operation rejects the first profile element for the interface because it refers to the wrong version numbers.
   b. In the next profile element, the operation finds the correct version numbers (`2.0`). The search proceeds to the associated server entry, `/.:/LandS/Cal_host_4`.

2. The search ends with the indicated server entry, where the binding information requested by the client resides:

   `/.:/LandS/Cal_host_4 contents:`

   *calendar-if-uuid*,`2.0`

   *binding-information*

**Example 2: Using a Default Profile for Importing an Interface**

**Target Interface:** Statistics V1.0

1. The search for binding information associated with Statistics V1.0 starts with the entry `molly_o_profile`. However, the profile lacks any elements for the interface. Therefore, the search reaches the default profile element, which provides the entry name for the default profile, `/.:/cell-profile`:

   `/.:/LandS/anthro/molly_o_profile contents:`

   *calendar-if-uuid*,`1.0   /.:/LandS/C_host_7   0` *calendar-if-uuid*,2.0   /.:/LandS/C_host_4   1

*bulletin_board-if-uuid*,2.0   /.:/LandS/bb_grp   2

*bulletin_board-if-uuid*,2.1   /.:/LandS/bb_grp   3

*Zork-if-uuid*,1.0   /.:/Eng/Zork_host_2   0

00000000-0000-0000-0000-000000000000,0.0   /.:/cell-profile   0

2. The search continues to the indicated default profile, **/.:/cell-profile**, which contains a profile element for the requested Statistics 1.0 interface:

   ```
   /.:/LandS/cell-profile contents:
   ```

   .
   .
   .
   *Statistics-if-uuid*,1.0   /.:/LandS/Stats_host_6   0
   .
   .
   .

3. The search ends at the indicated server entry, `/.:/LandS/Stats_host_6`, where a server address for the requested interface resides:

   ```
   /.:/LandS/Stats_host_6 contents:
   ```

   *Statistics-if-uuid,*1.0

   *binding-information*

**Example 3: Importing an Interface and an Object**

**Target Interface:** Print Server V2.1

**Target Object:** Laser Printer Print Queue

1. The search starts with the entry `/.:/Bldg/Print_queue_grp`, which contains the entry names of several server entries that advertise the `Print_server` interface and the object UUID of a given `Laser_printer` print queue. The search begins by randomly selecting a member name. In this instance, the search selects `/.:/Bldg/Print_server_host_3`:

   ```
   /.:/Bldg/Print_queue_grp contents:

      /.:/Bldg/Print_server_host_3
      /.:/Bldg/Print_server_host_7
      /.:/Bldg/Print_server_host_9
   ```

2. The search continues with the `/.:/Bldg/Print_server_host_3` entry. There it finds the requested Version 2.1 of the `Print_server` interface. However, the search continues, because the entry lacks the object UUID of the requested `Laser_printer` queue:

   ```
   /.:/Bldg/Print_server_host_3 contents:
   ```

   *print_server-if-uuid,*2.1

*binding-information*

*line_printer_queue-object-uuid*

3. The search returns to the previous entry, `/.:/Bldg/Print_queue_grp`, to select another entry name, in this instance, `/.:/Bldg/Print_server_host_9`:

   ```
   /.:/Bldg/Print_queue_grp contents:

      /.:/Bldg/Print_server_host_3
      /.:/Bldg/Print_server_host_7
      /.:/Bldg/Print_server_host_9
   ```

4. The search selects the `/.:/Bldg/Print_server_host_9` entry. This entry contains both a server address for the requested Version 2.1 of the interface and the requested object UUID of the `Laser_printer` queue:

   ```
   /.:/Bldg/Print_server_host_9 contents:
   ```

   *print_server-if-uuid*, `2.1`

   *binding-information*

   *laser_printer_queue-object-uuid*

   The search returns binding information from this entry to the client.

**Expiration Age of a Local Copy of Directory Service Data:** To prevent the unnecessary access of a namespace, previously requested directory service data is sometimes stored on the system where the request originated. A local copy of directory service data (also called the CDS cache) which is maintained by the CDS Clerk daemon, is not automatically updated at each request. Automatic updating of the local copy occurs only when it exceeds its expiration age. The *expiration age* is the amount of time that a local copy of directory service data from an NSI attribute can remain unchanged before a request from an RPC application for the attribute requires updating of the local copy. When an RPC application begins running, the RPC runtime randomly specifies a value between 8 and 12 hours as the default expiration age for that instance of the application. Most applications use only this default expiration age, which is global to the application.

An expiration age is used by an NSI next operation, which reads data from directory service attributes. For a given search or inquire operation, you can override the default expiration age by calling the **rpc_ns_mgmt_handle_set_exp_age()** routine after the operation's begin routine. Specifying a low default age will result in increased network updates among the name servers in your cell. Because this activity will adversely affect the performance of all network traffic, use the default whenever possible. If you must override the default age, specify a number high enough to avoid frequent updates of local data.

An NSI next operation usually starts by looking for a local copy of the attribute data being requested by an application. In the absence of a local copy, the NSI next operation creates one with fresh attribute data from the namespace. If a local copy already exists, the operation compares its actual age to the expiration age used by the application. If the actual age exceeds the expiration age, the operation automatically tries to update the local copy with fresh attribute data. If updating is impossible, the old local data remains in place and the NSI next operation fails, returning the **rpc_s_name_service_unavailable** status code.

**Note:** If your client successfully imports the bindings for its server but receives a communications failure status, **comm_status**, when it attempts the RPC, it is possible that the server has been transferred to another machine and the local directory service data is no longer current.

You can handle this situation in your client's error recovery code by disabling the local directory service cache by setting the expiration age to 0 (zero). You can use **rpc_ns_mgmt_set_exp_age()** to modify the global expiration age that the application uses, or you can use **rpc_ns_mgmt_handle_set_exp_age()** to set the expiration age for a specific name service handle in the case of a multithreaded application. By setting the expiration age to 0, your application accesses the directory service data directly from the CDS server. See *z/OS DCE Application Development Reference* for details on using the above APIs.

As stated above, setting this expiration age to a low value will result in increased network traffic and will deteriorate your application's performance.

## Strategies for Using Directory Service Entries

When developing an RPC application, decide how an application will use the namespace and design your application accordingly. The following subsections discuss issues associated with how servers use different types of directory service entries.

**Using Server Entries:** Each server entry requires a unique cell-relative entry name. (See "Guidelines for Constructing Names of Directory Service Entries" on page 104 for the guidelines for constructing directory name service entries.) If a server adheres to a simple and consistent arrangement of server entries, use server initialization code to automatically generate a name for each server entry and to ensure that the entry exists. However, some servers will need to obtain the entry name of a server entry from an external source such as a command-line argument or a local database belonging to the application.

Identical servers on different hosts should be identified by separate server entries. For example, if a server offering the calendar service runs on two hosts, JULES and VERNE, one server entry is necessary for the server on JULES and another is necessary for the server on VERNE.

Some applications, such as a process-control application, require only one server instance per system. Many applications, however, can accommodate multiple server instances on a system. When multiple instances of a server run simultaneously on a single system, all instances on a host can use a single server entry, or every instance can use separate server entries. Alternatively, the instances can be classified into subsets, with a separate server entry. A client importing from a shared server entry cannot distinguish among the server instances that export to the entry. Therefore, the recommended strategy for a server on a given system depends on which server instances are viewed by clients as interchangeable entities and which are viewed as unique entities as follows:

- Interchangeable server instances

  When clients consider all the server instances on a host as equivalent alternatives, all of the instances can (and should) share a server entry. For example, multiple instances of the calendar service running on host JULES can all export to the `calendar_JULES` entry.

- Unique server instances

  A unique server instance possesses a significant difference from other instances of the same host. Unique server instances require separate server entries. Each server instance must export unique information to its own server entry; this unique information can be either a server-specific, well-known endpoint or an object UUID belonging exclusively to the one server instance.

Before exporting, each server instance must acquire the entry name of its server entry from an external source. When a unique server instance stops running, its server entry becomes available. To reuse an available server entry for a new instance of that server, provide the existing entry's name for the new server instance to use with the export operation. If any existing server entries are unavailable, a new server instance requires a new server entry name.

**Note:** For a discussion of removing the binding information from its server entry, see **rpc_ns_binding_unexport** in *z/OS DCE Application Development Reference*.

**Using Groups:** When a server is first installed on a system, the server or the installer creates one or more server entries for the server. When installing the first instance of the server within a cell, the installer usually creates one or more groups for the application. Note that this notion of group, associated with the CDS namespace, is not the same as the notion of a POSIX user group used in the UNIX System Services Shell. For any application, the local system and directory service administrators can create site-specific groups whose members are server entries, groups, or both. Typically, a server adds a server entry to at least one group.

Design decisions for defining groups may reflect a number of possible factors. Typical factors that help define effective groups include the proximity of services or resources to clients, the types of any resources offered by servers, the uses of UUIDs, and the types of users that require a specific server.

For example, for a print server, proximity to the clients and the type of supported file formats are both relevant. These factors may affect print servers as follows:

- Proximity

  If the proximity of a server is important to clients, assign servers to groups according to their locations. For example, print servers that are located together can use their own group. (For example, print servers in building 1 use the group `bldg_1_print_servers`.) Each server instance can add its own entry to the group, or a system administrator can add server entries using the RPC control program.

  To select randomly among servers in a given location, a client imports a binding using the name of a group that corresponds to those servers (or of a profile that refers to that group).

  **Note:** If proximity is the key factor in selecting among servers, name each server entry for the server's location; for example, `bldg_1_pole_27_print_server`.

- Object types

  If accessing specific classes of resources is important to clients, you can group server instances based on the type of object they offer.

  For servers that advertise resources in server entries, groups often use subset server entries according to the resources they advertise. For example, print servers can be grouped according to supported file formats. In this case, an administrator creates a group entry for each file format, for example, `post_printers`, `sixel_printers`, and `ascii_printers`. Each print server entry is a member of one or more groups.

  Users that specify a group for a file format must find the printer that processes the print command. To help the user find the printer, the client can obtain the name of the server entry that supplied the server binding information (by calling **rpc_ns_binding_inq_entry_name()**) and then display the name for the user. If the server entry name indicates the location of the print server (for example, `floor_3_room_45A_print_server`), the user can find the printer.

An application can set up groups according to different factors for different purposes. For example, the print server application can set up groups of neighboring print servers and a group of print servers for each of the file formats. The same server is a member of at least one group of each kind. Clients require users to specify the name of a directory service entry as a command-line argument of remote print commands. The user specifies the name of the appropriate group.

**Note:** If a user wants a specific print server and knows the name of its server entry, the user can specify that name to the client instead of a group.

**Using Profiles:** Profiles are tools for managing NSI searches (performed by **rpc_ns_binding_import_next** or **rpc_ns_binding_lookup_next** operations). Often profiles are set up as public profiles for the users of a particular environment, such as a directory service cell, a system, a specific application, or an organization. For example, the administrator of the local directory service cell should set up a cell profile for all RPC applications using the cell, and the administrator of each system in the distributed computing environment should set up a system profile for local servers.

For each application, a directory service administrator or the owner of an application should add profile elements to the public profiles that serve the general population, for example, a cell profile, a system profile, or a profile of an organization. Each profile element associates a *profile member* (represented in the member field of an element as the global name of a directory service entry) with an interface identifier, access priority, and optional annotation. A candidate for membership in a cell profile is a group or another profile, for example, a group that refers, directly or indirectly, to the servers of an application installed in the local cell or an application-specific profile.

An application can benefit from an application-specific profile. For example, an administrator at a specific location, such as a company's regional headquarters, can assign priorities to profile elements based on the proximity of servers to the headquarters, as illustrated in Figure 29. When assigning profile element priorities, use 1 through 7 and avoid using priority 0 as the highest priority. See page 102 for a description of why you should adhere to this convention.

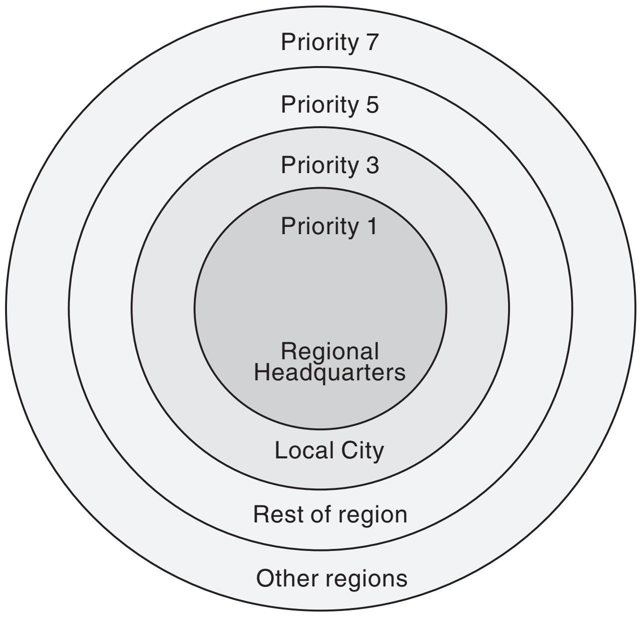


*Figure 29. Priorities Assigned on Proximity of Members*

An individual user can have a personalized *user profile* that contains elements for interfaces the user uses regularly and a default element that specifies a public profile, such as the cell profile, as the default profile. NSI searches use the default profile when a client needs an RPC interface that lacks an element in the user profile.

## The Service Model for Defining Servers

The NSI operations accommodate two distinct models for defining servers: the service model and the resource model. These models express different views of how clients use servers and how servers can present themselves in the directory service database. The models are not mutually exclusive, and an application may need to use both models to meet diverse goals. By evaluating these models before designing an RPC application, you can make informed decisions about whether and how to use object UUIDs, how many server entries to use per server, how to distinguish among instances of a server on a system, whether and how to use groups or profiles or both, and so forth. The two models are the Service model and the Resource model.

The Service model views a server exclusively as a distributed service composed of one or more application-defined interfaces that meet a common goal independently of specific resources. The service model is used by applications whose servers offer identical services and whose clients do not request an RPC resource when importing an interface. Often, with the service model, all the server instances of an application are equivalent and are viewed as interchangeable. However, the service model can accommodate applications that view each server instance as unique. Because the implications of whether server instances are viewed as interchangeable or unique are significant, the following subsections address these alternatives separately.

**Interchangeable Server Instances:** With the service model, servers offer an identical service that operates the same way on all host systems. For example, an application that uses the service model is a collection of equivalent print servers that support an identical set of file formats installed on printers in a single location. The print servers in any location can be segregated from print servers elsewhere by using a location-specific group.

Figure 30 on page 118 shows interchangeable print servers offering an identical print service on different hosts. To access this service, clients request the Print V1.0 interface and specify the nil object UUID. In this illustration, the starting entry for the NSI search is a group corresponding to local print servers. A client may be able to reach this print server group by starting from a profile or another group. To simplify the illustrations of the usage models, the contents of server entries are represented without listing any binding information.

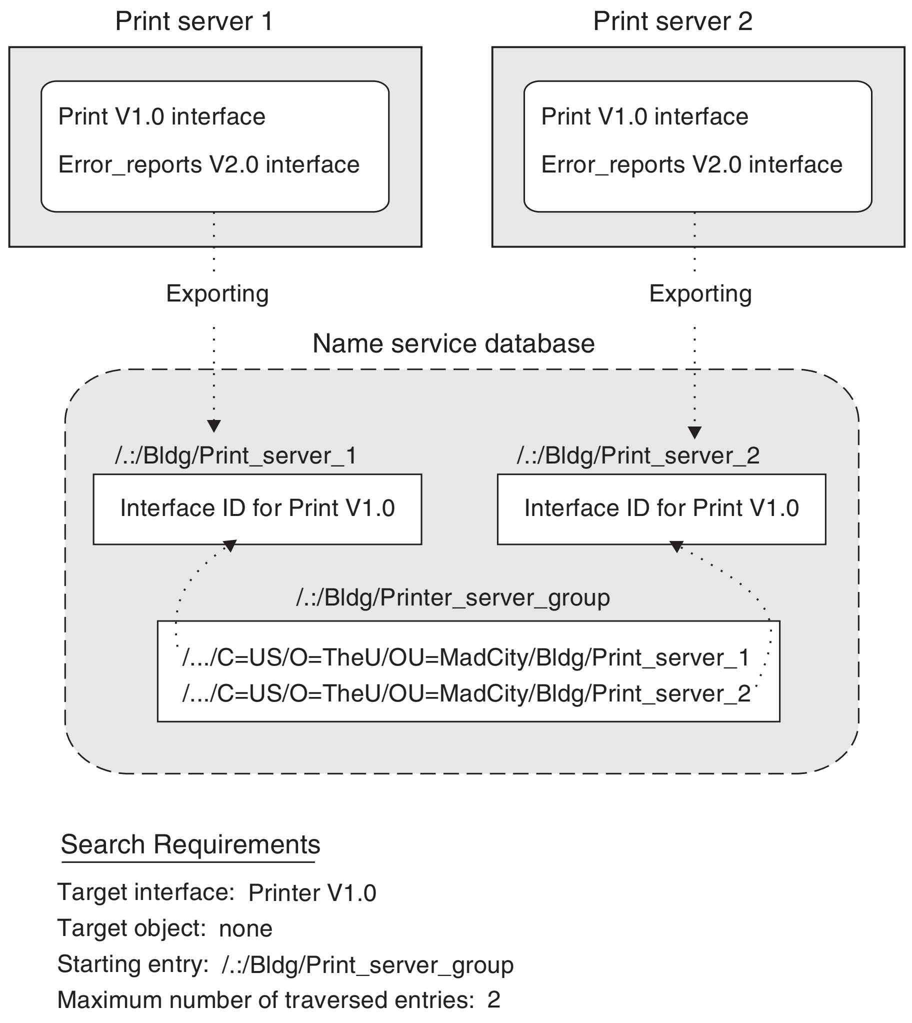

Print server 1
Print server 2
Print V1.0 interface
Error_reports V2.0 interface
Print V1.0 interface
Error_reports V2.0 interface
Exporting
Exporting
Name service database
/.:/Bldg/Print_server_1
Interface ID for Print V1.0
/.:/Bldg/Print_server_2
Interface ID for Print V1.0
/.:/Bldg/Printer_server_group
/.../C=US/O=TheU/OU=MadCity/Bldg/Print_server_1
/.../C=US/O=TheU/OU=MadCity/Bldg/Print_server_2
Search Requirements
Target interface: Printer V1.0
Target object: none
Starting entry: /.:/Bldg/Print_server_group
Maximum number of traversed entries: 2

*Figure 30. Service Model: Interchangeable Instances on Two Hosts*

The number of entries traversed by a search operation is unrelated to the number of binding handles it returns.

Figure 31 on page 119 shows interchangeable server instances offering an identical statistics service on a single host. To access this service, clients request the Statistics V1.0 interface and specify the nil object UUID. The starting entry for the NSI search is a group corresponding to local servers that offer the service (or a profile that refers to that group).

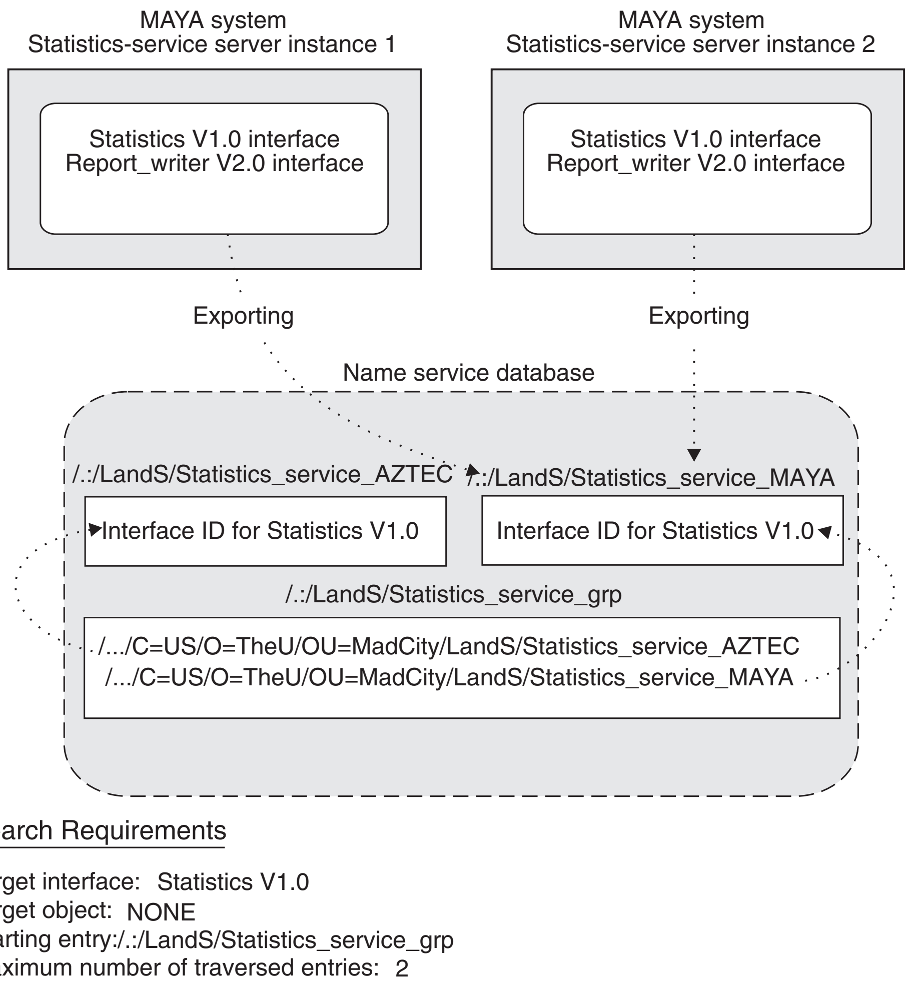

MAYA system
Statistics-service server instance 1
MAYA system
Statistics-service server instance 2
Statistics V1.0 interface
Report_writer V2.0 interface
Statistics V1.0 interface
Report_writer V2.0 interface
Exporting
Exporting
Name service database
/.:/LandS/Statistics_service_AZTEC
/.:/LandS/Statistics_service_MAYA
Interface ID for Statistics V1.0
Interface ID for Statistics V1.0
/.:/LandS/Statistics_service_grp
/.../C=US/O=TheU/OU=MadCity/LandS/Statistics_service_AZTEC
/.../C=US/O=TheU/OU=MadCity/LandS/Statistics_service_MAYA
Search Requirements
Target interface: Statistics V1.0
Target object: NONE
Starting entry:/.:/LandS/Statistics_service_grp
Maximum number of traversed entries: 2

*Figure 31. Service Model: Interchangeable Instances on One Host*

If an application with interchangeable server instances uses the connectionless RPC protocol, the default behavior of the endpoint map service is to always return the endpoint from the first map element for that set of server instances. To avoid having clients using only one of the instances, before making a remote procedure call to the server, each client must inquire for an endpoint. For a random selection, a client calls the **rpc_ep_resolve_binding()** routine. Alternatively, a client can call the **rpc_mgmt_ep_elt_inq_...()** routines to obtain all the map elements for compatible server instances, and then use an application-specific selection algorithm to select one of the returned elements.

**Distinct Server Instances on a Single Host:** With the service model, when multiple server instances on a given host are somehow unique, each instance must export to a separate server entry. The exported binding information must contain one or more instance-specific well-known endpoints or an object UUID that represents an instance (that is, an instance UUID). Well-known endpoints and instance UUIDs are used under the following circumstances:

- Well-known endpoints

  An instance-specific, *well-known endpoint* must be provided to a server instance as part of its installation, for example, as a command-line argument. Before calling the export operation, the server instance tells the RPC runtime to use each of its well-known endpoints; it does this by calling **rpc_server_use_protseq_ep()**. The runtime includes these endpoints in the instance's binding information, which the runtime makes available to the instance through a list of server binding handles.

The server instance uses this list of binding handles to export its binding information, including the well-known endpoints. The server also uses this list of binding handles to register its well-known endpoint with the local endpoint map; it does this by calling **rpc_ep_register()** or **rpc_ep_register_no_replace()**). Remote calls made using an imported well-known endpoint from a server entry are guaranteed by the RPC runtime to go only to the server instance that exported the endpoint to that entry.

**Note:** Only one server instance per system can use a well-known endpoint obtained from a given interface specification.

- Instance UUID

  Create an instance UUID only for a new server entry. Generating a new instance UUID each time a server instance exports its bindings to a server entry will result in many instance UUIDs that are difficult to manage. It may also affect performance as new instance UUIDs are constantly added to server entries. If a new server instance inherits a currently unused server entry left behind by an earlier instance, before exporting bindings, the new server instance should inquire for an instance UUID in the server entry; this is done by calling the **rpc_ns_entry_object_inq_**{**begin**,**next**, **done**}() routines. If the inherited entry contains an instance UUID, the server uses it for an instance UUID, rather than creating and exporting a new instance UUID. If an inherited entry lacks an instance UUID, the server must create a UUID and export it to the server entry. Note that every server instance must register its instance UUID along with its endpoints in the local endpoint map.

Figure 32 on page 121 shows distinct instances of a statistics-service server on the same host. Each server instance uses an instance UUID to identify itself to clients. The instance UUID is the only object UUID a server instance exports to its server entry. Starting at the statistics-service group, clients import the statistics interface.

After finding a server entry with compatible binding information for the statistics interface, the import operation returns an instance UUID along with binding information. Every RPC made with that binding information goes to the server instance that exported the instance UUID.

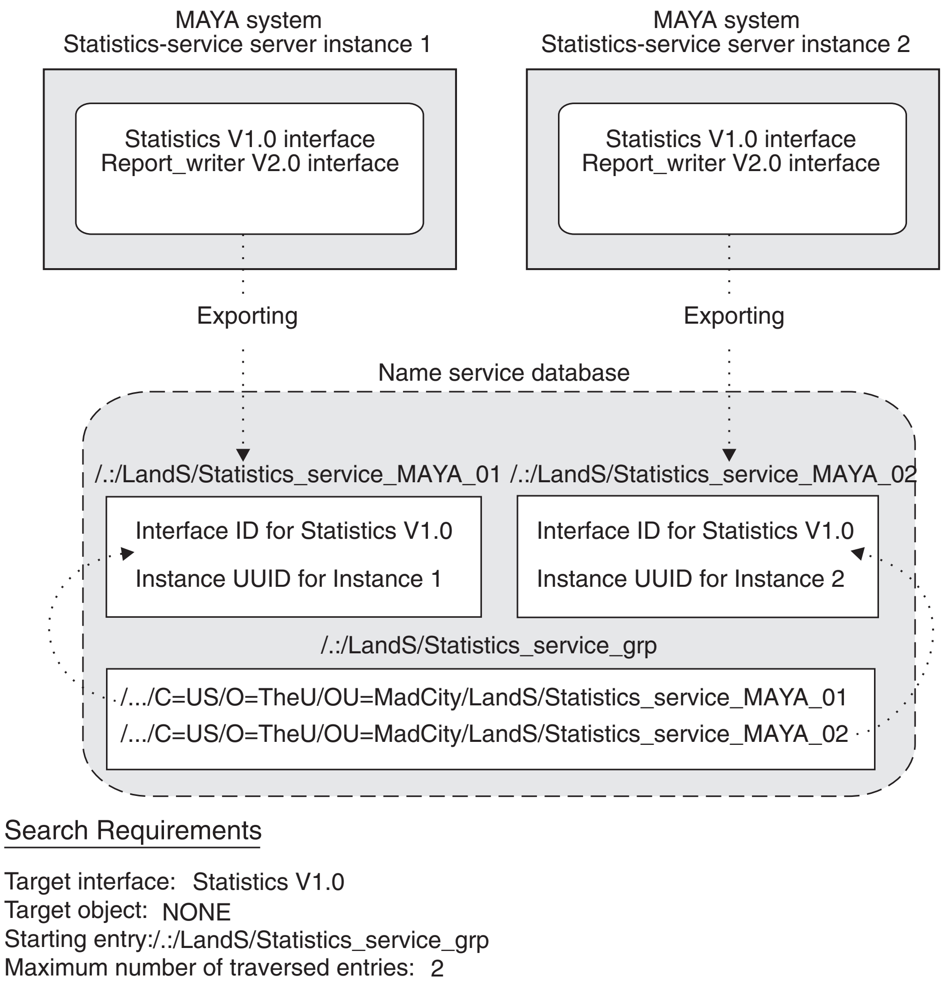

MAYA system
Statistics-service server instance 1
MAYA system
Statistics-service server instance 2
Statistics V1.0 interface
Report_writer V2.0 interface
Statistics V1.0 interface
Report_writer V2.0 interface
Exporting
Exporting
Name service database
/.:/LandS/Statistics_service_MAYA_01
/.:/LandS/Statistics_service_MAYA_02
Interface ID for Statistics V1.0
Instance UUID for Instance 1
Interface ID for Statistics V1.0
Instance UUID for Instance 2
/.:/LandS/Statistics_service_grp
/.../C=US/O=TheU/OU=MadCity/LandS/Statistics_service_MAYA_01
/.../C=US/O=TheU/OU=MadCity/LandS/Statistics_service_MAYA_02
Search Requirements
Target interface: Statistics V1.0
Target object: NONE
Starting entry:/.:/LandS/Statistics_service_grp
Maximum number of traversed entries: 2

*Figure 32. Service Model: Distinct Instances on One Host*

## The Resource Model for Defining Servers

The Resource model views servers and clients as manipulating resources. A server and its clients use object UUIDs to identify specific resources. With the resource model, any resource an application's servers and clients manipulate using an object UUID is considered an *RPC resource*. Typically, an RPC resource is a physical resource such as a database. An RPC resource can also be abstract, for example, a print format such as ASCII. An application that uses the resource model for one context may use the service model for another.

Applications in which a client requests a server to operate on a particular RPC resource use the resource model. Each server accesses one or more resources, such as print servers or databases. Applications use object UUIDs to refer to resources as follows:

1. Servers offer resources by assigning an object UUID to each specific resource.
2. Clients obtain those object UUIDs and use them to learn about a server that offers a given resource. Clients that use the resource model cannot use automatic binding.
3. When making an RPCcall, a client requests a resource by passing its UUID as part of the server binding information.

Each RPC resource or type of resource requires its own object UUID. A calendar server, for example, may require a distinct UUID to identify each calendar.

RPC interfaces can be defined to operate with different types of resources and can be carried out separately for each type, for example, a print server application that supports sixel, ASCII, and PostScript file formats. When using different implementations of an interface (different managers), servers must associate the object UUID of a resource, such as an ASCII file format and its manager, by assigning them a single type UUID. To request the resource, a client specifies its object UUID in the server binding information. When a print server receives the RPC, it looks up the corresponding type UUID and selects the associated manager.

**Choosing between Service and Resource Models:** When developing an RPC application, you need to decide whether to use object UUIDs to identify RPC resources and, if so, what sorts of resources receive UUIDs that servers export to the namespace. When making these decisions, consider the following questions:

- Will users need to select a server entry from the namespace based on what object UUIDs the entry contains (and what the client needs)?

  If yes, then a client must specify an object UUID to the import operation.
- Does the type of resource you are using last for a long time (months or years), so you can advertise object UUIDs efficiently in the namespace?

  The information kept in a namespace should be static or rarely change. For example, print queues are appropriate RPC resources. In contrast, quickly changing information, such as the jobs queued for the printer, owners of the jobs, or the time the job was added to the queue, should not be viewed as RPC resources. Such short-lived data may be viewed as local objects, which are stored and managed at a specific server. Programming with local objects is in the area of regular object-oriented programming and is independent of an application's use of RPC resources.
- Is the number of objects belonging to the type of resource bounded (to avoid placing high demands on the directory service)? If not, use the service model.
- Will the server use an interface for different types of a resource, such as different forms of calendar databases or different types of queues?

  If yes, then the server must classify objects into types. For each type, the server generates a non-nil UUID for the type UUID, sets the type UUID for every object of the type, and specifies that type as the manager type when registering the interface. When making an RPC to the interface, a client must supply an object UUID to specify an RPC resource.
- Is control over specific resources an important factor for distinguishing among server instances on a host?

  If yes, then each server must generate an object UUID for each of its resources.

For some applications, such as those accessing a database that many people use, shared access to one or more objects may be essential. However, not all objects accommodate such shared access. Generally, use the service model whenever possible because it facilitates a simpler client implementation.

**Using Objects and Groups Together:** Servers can associate object UUIDs with a group. Each server exports one or more object UUIDs (without exporting any binding information) to the directory service entry of the group; this involves specifying the NULL interface identifier to the export operation along with the list of object UUIDs. The object UUIDs reside in the directory service entry of the group. If a server stops offering an advertised object, a server must unexport its object UUID from the group entry to keep its object list up to date.

Clients use objects in a group entry as follows:

1. The client inquires for an object UUID from the group entry by calling the **rpc_ns_entry_object_inq_**{**begin**,**next**,**done**} () routines. This routine selects one object UUID at random and returns it to the client.
2. The client imports binding information for the returned object UUID (and the interface of the called remote procedure), specifying the group for the start of the search.
3. The import operation returns a binding handle that refers to the requested object UUID and binding information for a server that offers the corresponding object.
4. The client issues the RPC using that binding handle.
5. The server looks up the type of the requested object.
6. The server assigns the RPC to the manager that carries out the called remote procedure for that type of object.

**System-Specific Applications:** For some applications, the clients need to import an RPC resource that belongs to a specific system, and the clients can specify a server entry name to learn about a server on that system. For example, a process server that allows clients to monitor and control processes on a remote machine is useful only to that machine. Figure 33 on page 124 illustrates this type of system-specific interpretation of the resource model.

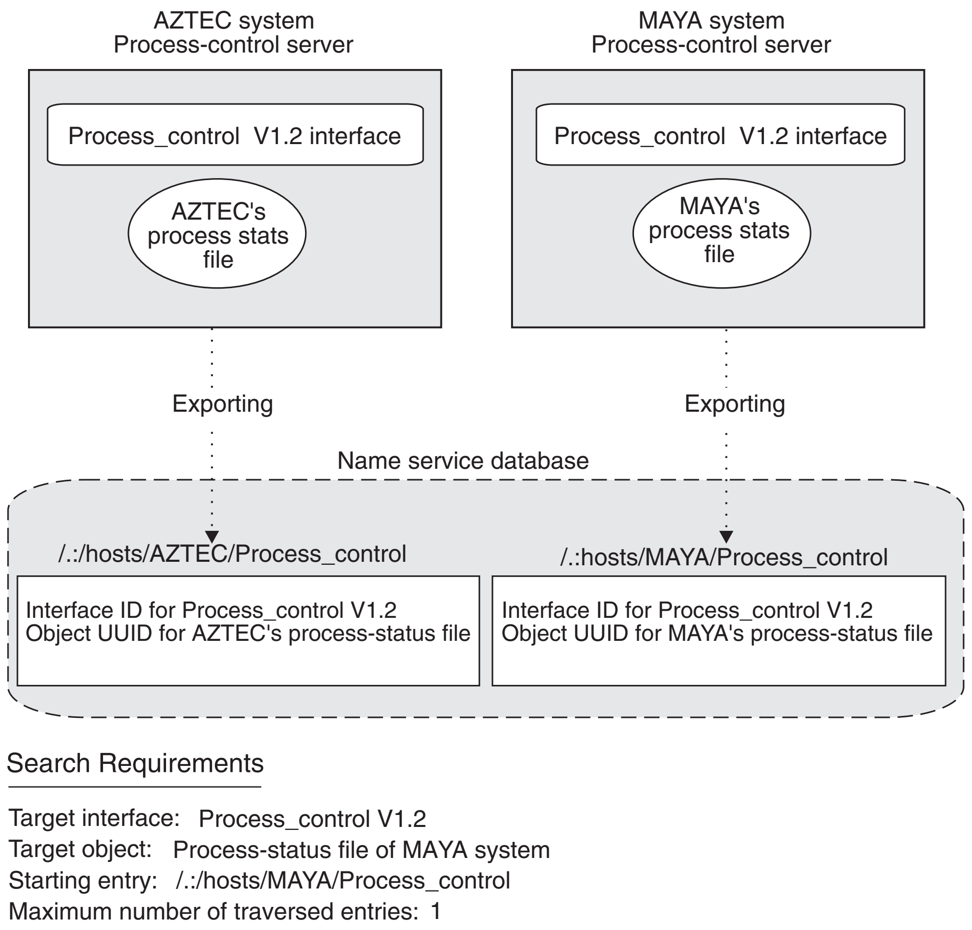

AZTEC system
Process-control server
Process_control V1.2 interface
AZTEC's process stats file
MAYA system
Process-control server
Process_control V1.2 interface
MAYA's process stats file
Exporting
Exporting
Name service database
/.:/hosts/AZTEC/Process_control
Interface ID for Process_control V1.2
Object UUID for AZTEC's process-status file
/.:hosts/MAYA/Process_control
Interface ID for Process_control V1.2
Object UUID for MAYA's process-status file
Search Requirements
Target interface: Process_control V1.2
Target object: Process-status file of MAYA system
Starting entry: /.:/hosts/MAYA/Process_control
Maximum number of traversed entries: 1

*Figure 33. Resource Model: A System-Specific Application*

Because clients usually find a system-specific server by specifying its server entry to the import operation, groups are usually not part of the NSI search path for system-specific applications. Groups are a management tool for such applications. A group containing the names of the server entries of all the current servers can act as an accounting database. Also, a group for the servers on each set of related systems, such as the members of a LAN or an administrative grouping, permits a client to sequentially use the application on every system in the set. An application with system-specific servers should **not** use profiles.

**Exporting Multiple Object UUIDs to a Single Server Entry:** Often a single server offers more than one resource or several types of resources. Where a server instance has a large number of object UUIDs, the application should place multiple object UUIDs into a single server entry. Typically, an application places all of its object UUIDs into one server entry. However, you may need to segregate them into several server entries according to factors such as object type, location, or who uses the different types of objects. When you are dividing resources, try to assign each resource to a single set so that its object UUID is exported to only one server entry. Figure 34 on page 125 illustrates a single server entry implementation for each server for the resource model.

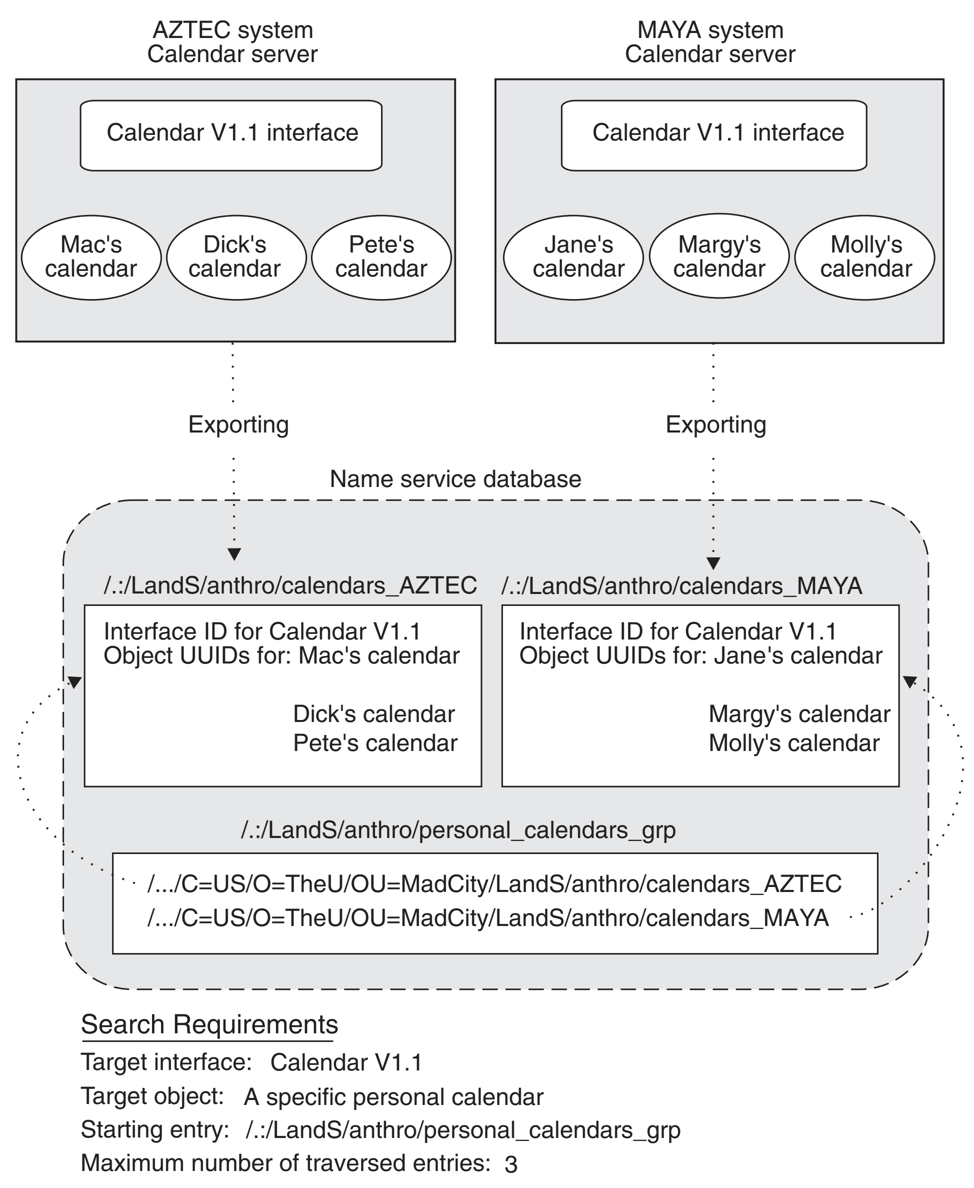

AZTEC system
Calendar server
Calendar V1.1 interface
Mac's calendar
Dick's calendar
Pete's calendar
MAYA system
Calendar server
Calendar V1.1 interface
Jane's calendar
Margy's calendar
Molly's calendar
Exporting
Exporting
Name service database
/.:/LandS/anthro/calendars_AZTEC
Interface ID for Calendar V1.1
Object UUIDs for: Mac's calendar
Dick's calendar
Pete's calendar
/.:/LandS/anthro/calendars_MAYA
Interface ID for Calendar V1.1
Object UUIDs for: Jane's calendar
Margy's calendar
Molly's calendar
/.:/LandS/anthro/personal_calendars_grp
/.../C=US/O=TheU/OU=MadCity/LandS/anthro/calendars_AZTEC
/.../C=US/O=TheU/OU=MadCity/LandS/anthro/calendars_MAYA
Search Requirements
Target interface: Calendar V1.1
Target object: A specific personal calendar
Starting entry: /.:/LandS/anthro/personal_calendars_grp
Maximum number of traversed entries: 3

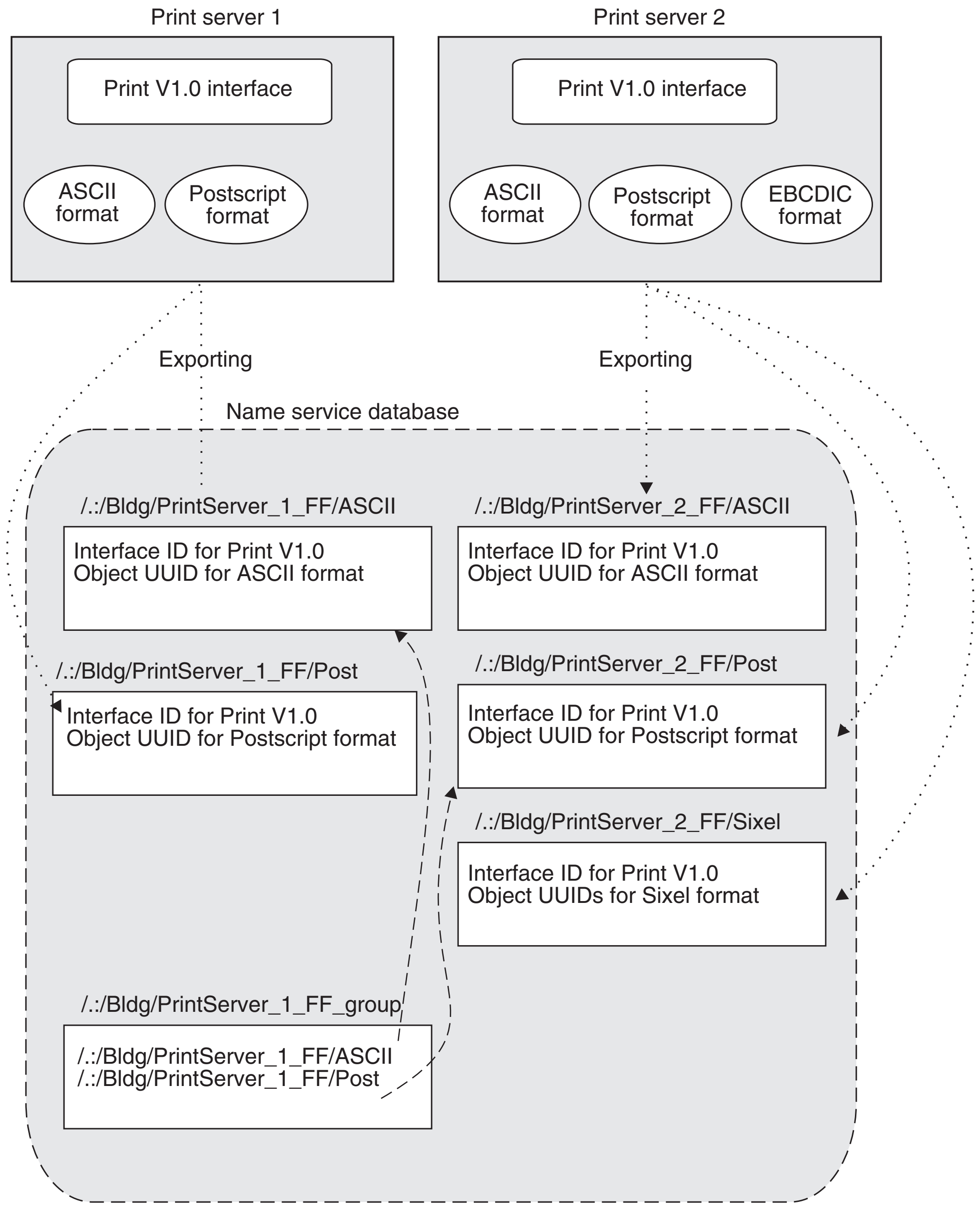
*Figure 34. Resource Model: A Single Server Entry for Each Server*

**Exporting Every Object UUID to a Separate Server Entry:** For some applications, exporting each object UUID to a separate server entry is a practical strategy. To avoid excessive demands on directory service resources, however, this strategy requires that the set of objects remain small. Applications with many RPC resources should have each server create a single server entry for itself and export the object UUIDs of the resources it offers to that server entry. For example, an application that accesses a different personal calendar for every member of an organization needs to avoid using a separate server entry for each calendar.

For some applications, however, you can use a separate server entry for each object UUID, for example, a print server application that supports a small number of file formats. Each server can create a separate server entry for each supported file format and export its object UUID to that server entry. The server entries for a file format are members of a distinct group.

To import binding information for a server that supports a required file format, a client specifies the nil UUID as the object UUID and the group for that format as the starting entry. The import operation selects a group member at random and goes to the corresponding server entry. Along with binding information, the operation returns the server's object UUID for the requested file format from the server entry. When the client issues an RPC to the server, the imported object UUID correctly identifies the file format the client needs. Figure 35 illustrates this use of object UUIDs.

.



Print server 1
Print V1.0 interface
ASCII format
Postscript format
Print server 2
Print V1.0 interface
ASCII format
Postscript format
EBCDIC format
Exporting
Exporting
Name service database
/.:/Bldg/PrintServer_1_FF/ASCII
Interface ID for Print V1.0
Object UUID for ASCII format
/.:/Bldg/PrintServer_2_FF/ASCII
Interface ID for Print V1.0
Object UUID for ASCII format
/.:/Bldg/PrintServer_1_FF/Post
Interface ID for Print V1.0
Object UUID for Postscript format
/.:/Bldg/PrintServer_2_FF/Post
Interface ID for Print V1.0
Object UUID for Postscript format
/.:/Bldg/PrintServer_2_FF/Sixel
Interface ID for Print V1.0
Object UUIDs for Sixel format
/.:/Bldg/PrintServer_1_FF_group
/.:/Bldg/PrintServer_1_FF/ASCII
/.:/Bldg/PrintServer_1_FF/Post

Search Requirements

Target interface: Print V1.0
Target object: ASCII file format (client specifies nil object UUID)
Starting entry: /.:/Bldg/ASCII_FF_group
Maximum number of traversed entries: 2

*Figure 35. Resource Model: A Separate Server Entry for Each Object*

Applications that use a separate entry for each object UUID need to use groups cautiously. Keeping groups small when clients are requesting a specific object is essential, because an NSI search looks up the group members in random order. Therefore, the members of a group form a localized flat NSI search path rather than the hierarchical path. Flat search paths are inefficient because the average search will look at half the members. Small groups are not a problem. For example, if a group contains only 4 members, each referring to a server entry that advertises a distinct set of RPC resources, the average number of server entries accessed in each search is 2 and the maximum is only 4. The larger the group, however, the more inefficient the resulting search path. For example, for a group containing 12 members, each referring to a server entry that advertises a distinct set of object UUIDs, the average search accesses 6 entries, and some searches access all 12 server entries.

Some RPC resources belong exclusively to a single server instance, for example, print queues. Some resources can be shared among server instances, for example, a file format or an airline-reservation database. For server instances on the same system, sharing a resource means that the object UUID cannot be used to distinguish between the two instances. For a print server, this is unlikely to be a problem, assuming that each printer runs only one instance of the print server. In contrast, an application with a widely accessed database, such as an airline reservation application, may need to ensure that clients can distinguish server instances from each other. An application can distinguish itself by supplying its clients with instance-specific information, for example, a well-known endpoint or an instance UUID.

**Note:** Multiple server instances that access the same set of resources can introduce concurrency control problems, such as two instances accessing a tape drive at the same time. Where the system provides concurrency control, servers may compete and have to wait for resources such as databases. Dealing with delayed access to shared resources may require an application-specific mechanism, such as queuing access requests.

# Chapter 8. DCE Data Representation Support Considerations

The prime function of DCE is to provide interoperability and access to data across heterogeneous systems. Naturally there are wide variations in the way data is represented on these systems. The challenge for a heterogeneous computing environment is to solve the variations between code pages and numeric representations among these systems.

A *code page* is the hexadecimal encoding of all available glyphs, and a *glyph* is the printed appearance of a character. Each code page is intended to serve one linguistic environment. z/OS systems use many Extended Binary Coded Decimal Interchange Code (EBCDIC) code pages. AIX and UNIX-based systems typically use American National Standard Code for Information Interchange (ASCII) code pages.

For numeric data, System 370 machines use the System 370 format to represent floating point and integer data.

## The DCE Model

The DCE model deals with the above problem of multiple code pages by making the z/OS machine appear to be like an ASCII machine to the rest of the DCE network. That is, it ensures that all outgoing character data is converted from the local code page to ASCII code page ISO8859-1 (IBM code page 819) *on the wire*. Conversely, it ensures that all incoming character data is converted from ASCII ISO8859-1 (on the wire) to the local code page on the z/OS machine.

For numeric data, the outgoing data is sent as is on the wire. The receiver of the numeric data converts this incoming numeric data to the local representation.

See Figure 36 on page 130 for the DCE model.

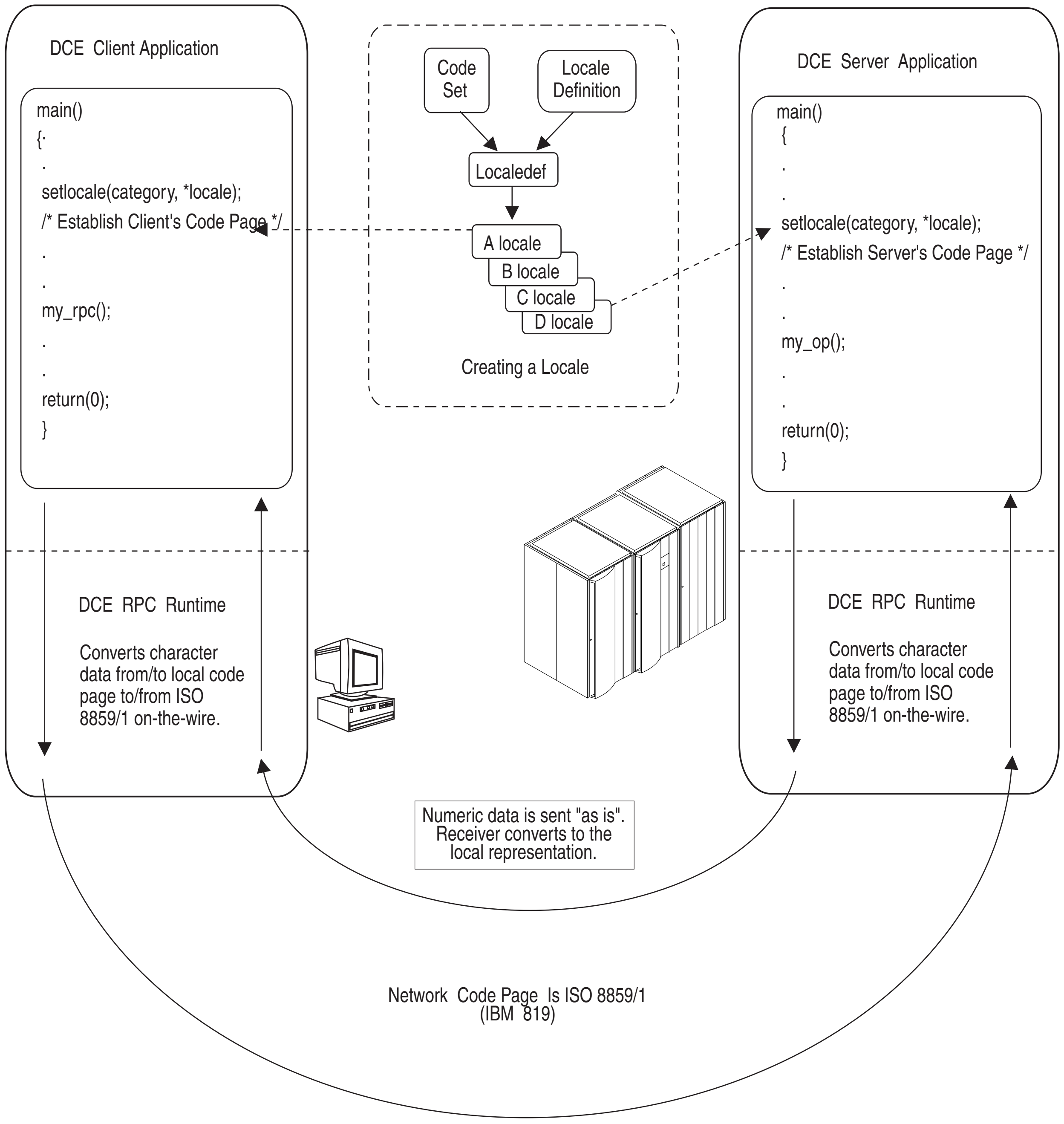

DCE Client Application
main()
{
.
setlocale(category, *locale);
/* Establish Client's Code Page */
.
.
my_rpc();
.
.
return(0);
}
Code Set
Locale Definition
Localedef
A locale
B locale
C locale
D locale
Creating a Locale
DCE Server Application
main()
{
.
.
setlocale(category, *locale);
/* Establish Server's Code Page */
.
.
my_op();
.
.
return(0);
}
DCE RPC Runtime
Converts character data from/to local code page to/from ISO 8859/1 on-the-wire.
DCE RPC Runtime
Converts character data from/to local code page to/from ISO 8859/1 on-the-wire.
Numeric data is sent "as is". Receiver converts to the local representation.
Network Code Page Is ISO 8859/1 (IBM 819)

*Figure 36. The DCE Model for Handling Multiple Code Pages*

## Data Type Considerations for Users

When developing DCE applications, you must consider the following data types:

- Floating-point data
- Integer data
- Single-byte character data
- Double-byte character data.

These are discussed in the following sections.

## Floating-Point Data

There are several data representation formats for floating single-precision and double-precision floating-point, such as:

- IEEE single and double floating-point
- VAX F_floating and G_floating formats
- Cray floating-point format
- IBM long and short formats.

If data formats are different, DCE RPC automatically converts the data at the receiver's end (*receiver-makes-right*) so that the arguments passed to the remote procedure, and the results returned to the caller are interpreted correctly. This conversion is done using the ***Network Data Representation*** (***NDR***) protocol that defines how the structured values supplied in a call to a remote interface are encoded into byte stream format for transmission on the wire. z/OS DCE always converts floating point data received from the network to IBM long and short format for the local z/OS system.

## Integer Data

Integer data can have four sizes:

| | |
|---|---|
| **small** | 8-bit integer (1 byte) |
| **short** | 16-bit integer (2 bytes) |
| **long** | 32-bit integer (3 bytes) |
| **hyper** | 64-bit integer (4 bytes) |

The NDR protocol represents signed integers in two's complement notation, and unsigned integers as unsigned binary numbers. The byte sequence of integer data is represented in two ways:

**big-endian format**
Consecutive bytes of the byte stream representation are ordered from the most significant byte to the least significant byte.

**little-endian format**
Consecutive bytes of the byte stream representation are ordered from the least significant byte to the most significant byte.

If the ordering of bytes is different between machines, DCE RPC automatically converts the data at the receiver's end (*receiver-makes-right*) to the local system representation.

Of the above data types, you must pay particular attention to both single-byte and double-byte character data.

## Character Data

To maintain interoperability with other DCE implementations, all outgoing character data is converted to the ASCII code page, ISO8859/1, for transmission on the wire while all incoming character data is converted to the local code page that you select using **setlocale()**. The RPC runtime automatically converts parameters marked as **char** to the code page associated with the current locale. "Establishing a Current Locale" on page 137 describes the process for establishing a code page in your application using **setlocale()**. A detailed explanation of the process is provided in the *z/OS C/C++ Programming Guide*, SC09-4765.

**Notes:**

1. When you define an IDL data type to char, make sure you put only single-byte character data in that data type.
2. If you have a structure which contains mixed data, for example, a structure containing integer, binary and character fields, and you define the structure data type as character, RPC converts the whole structure. This is clearly not the intention of the user and will typically result in a user error.

There are three levels of character data that you should consider in your DCE applications:

- EBCDIC variant characters
- POSIX Portable Character Set (PPCS) characters
- SAA Latin-1 characters outside the PPCS.

Figure 37 represents the *universe* of SAA Latin-1 characters. Within that universe is a subset of 95 characters that are defined in the PPCS. Within the PPCS there is a further subset of 13 variant characters whose hexadecimal representations vary across IBM EBCDIC systems.

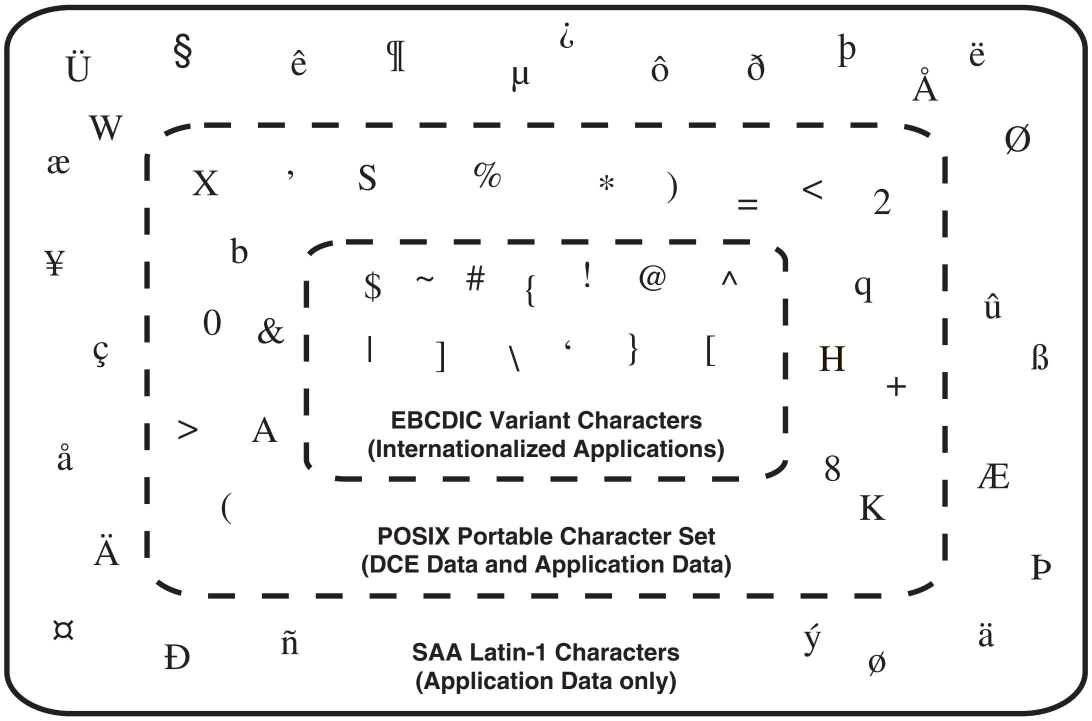


*Figure 37. How SAA Latin-1 Characters are Used in DCE*

**EBCDIC Variant Characters:** There are thirteen characters whose hexadecimal representations vary across IBM's EBCDIC code pages. Table 5 lists these thirteen SAA variant characters.

*Table 5 (Page 1 of 2). SAA Variant Characters*

| Variant Character | Symbol |
|---|---|
| backslash | \ |
| right brace | } |
| left brace | { |

*Table 5 (Page 2 of 2). SAA Variant Characters*

| Variant Character | Symbol |
|---|---|
| right bracket | ] |
| left bracket | [ |
| circumflex | ^ |
| tilde | ˜ |
| exclamation mark | ! |
| number sign | # |
| vertical bar | \| |
| dollar sign | $ |
| commercial at | @ |
| grave accent | ` |

You may have a variant character problem if you have data stored in a different code set than the code set that your application is running, and the data contains EBCDIC variant characters. If you do not perform any conversions, the variant character data that is stored will not display properly at a terminal. Similarly, if you update the data to a file, the data will get stored in the code set of your process, and the file data may become erroneous. To avoid this problem, you need to be aware of the code set of both your process and the file you are updating. The solution is to perform the proper code page conversions to normalize the data while reading it from the external medium to the application's code page. There is no such problem if your application process and data code page are identical. This code page conversion issue is purely a local system issue.

Consider Figure 38 where a user updates and reads data from a terminal using a database program.

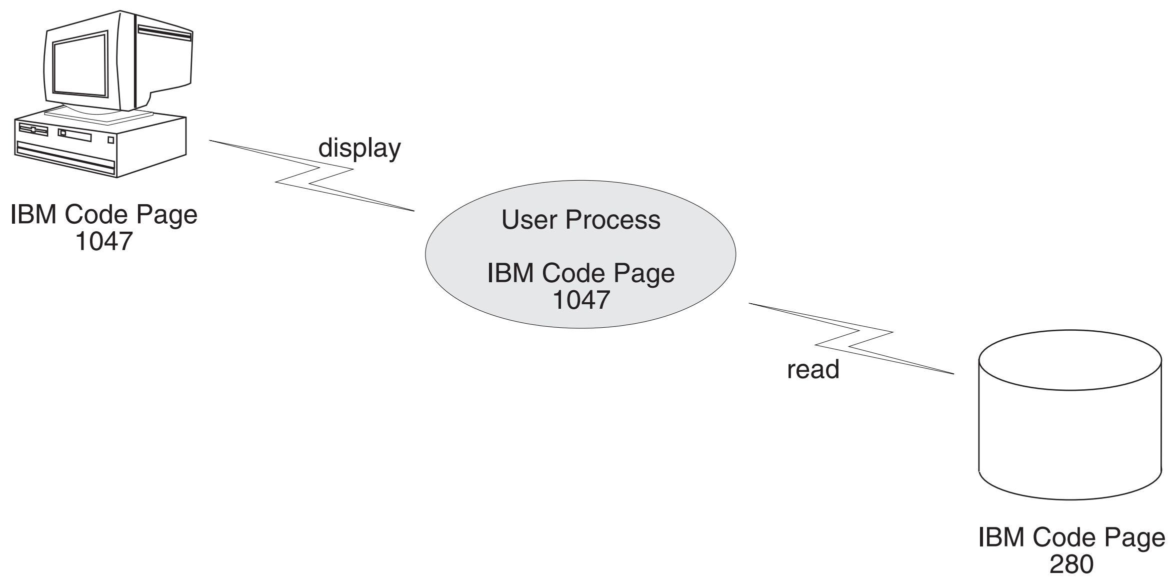


*Figure 38. Code Page Conversions*

In this scenario, the user must ensure that all the incoming or outgoing data is in EBCDIC code page 1047. If the user is retrieving data from a database that contains data of a different code page, say EBCDIC code page 280, the data is displayed incorrectly at the terminal if variant characters are used. To illustrate this problem, Table 6 on page 134 shows how some variant characters are represented in code page 1047 and code page 280.

*Table 6. Variant Characters — Code Point Representation*

| Character | EBCDIC Code Page 1047 Codepoint | EBCDIC Code Page 280 Codepoint |
|---|---|---|
| Left bracket ([) | `X'AD'` | `X'90'` |
| Right bracket (]) | `X'BD'` | `X'51'` |

If incoming data is in code page 280, the left and right square brackets will be erroneously displayed as Ý (Y acute) and ¨ (umlaut) unless the database application is internationalized, or the terminal is set up to handle code page 280.

**Eliminating the EBCDIC Character Variation Problem:** The EBCDIC variant character issue is resolved by using the **setlocale()** and LC_SYNTAX facilities of z/OS C/C++.

DCE internally uses the LC_SYNTAX solution provided by the z/OS C/C++ product. If your application has a sensitivity to any of the variant characters, and you expect to run it in multiple code pages, use the LC_SYNTAX capability of z/OS C/C++. See the *z/OS C/C++ Programming Guide*, SC09-4765 on this solution. To enable your DCE application for national language support, use **setlocale()** as shown in "Internationalization Considerations for DCE Applications" on page 136. If you are working with a single EBCDIC code page in your distributed environment in addition to ASCII, you do not have to address EBCDIC variant characters in your DCE applications.

See the z/OS C/C++ documentation for additional guidelines on writing code page independent code and how to write code that will be *exported*, that is, how to write code that will be run in a locale that uses a different code page than that used to initially develop the source. This information applies to DCE applications that are run on z/OS.

**POSIX Portable Character Set:** To avoid any network interoperability problems with your DCE applications whose client or server portion can run on other DCE platforms, you should restrict the data used by DCE daemons to the PPCS. The PPCS is defined by the POSIX 1003.2 standard. Data used by DCE typically includes data that is exported to the CDS namespace and stored in the Security Registry. This restriction does not apply to data used by your application; that is, data passed between your client and server. For example, if you use the German Ü (U umlaut) character in a CDS name or in a DCE principal name, your applications will fail. There is no assurance that the platform where the DCE daemons run can handle the non-PPCS characters; however, this does not mean that data exchanged between your client and server application cannot contain the U umlaut or other non-PPCS characters. Table 24 on page 537 lists the characters in the POSIX Portable Character Set, with their symbolic name and the graphic symbol for the character. Some of the characters, such as the hyphen, also have alternate symbolic names.

**Non-POSIX Portable Character Set Data:** You can use characters outside the PPCS for your application data only. You should not use characters outside the PPCS for data used by DCE daemons or your DCE applications may fail due to interoperability problems. Note that there is no validation or checking performed on this data, so you must ensure that data passed to the DCE daemons (non-application data) are within the PPCS.

**z/OS DCE Supported Code Pages:** With z/OS DCE, you can run your DCE applications using the code pages listed in 7.

*Table 7 (Page 1 of 2). Locales supported by z/OS DCE*

| Locale Name as setlocale() argument | Language | Country | Code Page |
|---|---|---|---|
| Da_DK.IBM-277 | Danish | Denmark | IBM-277 |
| Da_DK.IBM-1047 | Danish | Denmark | IBM-1047 |
| De_CH.IBM-500 | German | Switzerland | IBM-500 |
| De_CH.IBM-1047 | German | Switzerland | IBM-1047 |
| De_DE.IBM-273 | German | Germany | IBM-273 |
| De_DE.IBM-1047 | German | Germany | IBM-1047 |
| En_GB.IBM-285 | English | United Kingdom | IBM-285 |
| En_GB.IBM-1047 | English | United Kingdom | IBM-1047 |
| En_JP.IBM-1027 | English | Japan | IBM-1027 |
| En_US.IBM-037 | English | United States | IBM-037 |
| En_US.IBM-1047 | English | United States | IBM-1047 |
| Es_ES.IBM-284 | Spanish | Spain | IBM-284 |
| Es_ES.IBM-1047 | Spanish | Spain | IBM-1047 |
| Fi_FI.IBM-278 | Finnish | Finland | IBM-278 |
| Fi_FI.IBM-1047 | Finnish | Finland | IBM-1047 |
| Fr_BE.IBM-500 | French | Belgium | IBM-500 |
| Fr_BE.IBM-1047 | French | Belgium | IBM-1047 |
| Fr_CA.IBM-037 | French | Canada | IBM-037 |
| Fr_CA.IBM-1047 | French | Canada | IBM-1047 |
| Fr_CH.IBM-500 | French | Switzerland | IBM-500 |
| Fr_CH.IBM-1047 | French | Switzerland | IBM-1047 |
| Fr_FR.IBM-297 | French | France | IBM-297 |
| Fr_FR.IBM-1047 | French | France | IBM-1047 |
| Is_IS.IBM-871 | Iceland | Iceland | IBM-871 |
| Is_IS.IBM-1047 | Iceland | Iceland | IBM-1047 |
| It_IT.IBM-280 | Italian | Italy | IBM-280 |
| It_IT.IBM-1047 | Italian | Italy | IBM-1047 |
| Ja_JP.IBM-939 | Japanese | Japan | IBM-939 |
| Ja_JP.IBM-1027 | Japanese | Japan | IBM-1027 |
| Nl_BE.IBM-500 | Dutch | Belgium | IBM-500 |
| Nl_BE.IBM-1047 | Dutch | Belgium | IBM-1047 |
| Nl_NL.IBM-037 | Dutch | Netherlands | IBM-037 |
| Nl_NL.IBM-1047 | Dutch | Netherlands | IBM-1047 |
| No_NO.IBM-277 | Norwegian | Norway | IBM-277 |
| No_NO.IBM-1047 | Norwegian | Norway | IBM-1047 |

*Table 7 (Page 2 of 2). Locales supported by z/OS DCE*

| Locale Name as setlocale() argument | Language | Country | Code Page |
|---|---|---|---|
| Pt_PT.IBM-037 | Portuguese | Portugal | IBM-037 |
| Pt_PT.IBM-1047 | Portuguese | Portugal | IBM-1047 |
| Sv_SE.IBM-278 | Swedish | Sweden | IBM-278 |
| Sv_SE.IBM-1047 | Swedish | Sweden | IBM-1047 |

## Double-Byte Character Data

Your DCE applications may need to exchange Double-Byte Character Set (DBCS) data between the client and server. For example, your application may need to exchange data using an international character set such as Kanji. To handle double-byte character data, such as Kanji, use the **byte** attribute in the IDL file to declare the data type. Declaring data as **byte** prevents any data format conversion by the DCE RPC mechanism.

Since DBCS data is exchanged as a **byte** string, you have to correctly convert this data to your local code page when receiving data, and conversely, convert this data to the remote code page when transmitting it. This requires that you know the code page used on both your remote and local machine to properly make the conversions. z/OS DCE does not perform this conversion for data declared as **byte**.

For more information on using the **byte** attribute, see "The byte Type" on page 239.

## Internationalization Considerations for DCE Applications

Internationalization for DCE applications require that you consider all code page, translation, and data format issues when designing your application. Methods that help you internationalize your applications include using the **setlocale()** and **iconv()** routines.

**Localization and Code Page Conversions:** A locale is the definition of a user's environment that is dependent on language and cultural conventions. Locales are defined by POSIX standards. A locale consists of several categories that group information pertaining to an aspect of the language and cultural specific data. An example is LC_TIME, which specifies culturally specific rules regarding date and time. The following example shows how the sixth day of the ninth month of the year 1990 is represented in United States (US) and United Kingdom (UK) date formats:

| Country | Date Format |
|---|---|
| United States | 9/6/1990 (M/D/Y) |
| United Kingdom | 6/9/1990 (D/M/Y) |

Typically, a locale is made up of eight categories as follows:

| Category | Purpose |
|---|---|
| LC_COLLATE | Affects the behavior of the `strcoll()` and `strxfrm()` C functions. |
| LC_CTYPE | Affects the behavior of the character handling functions. |
| LC_MESSAGES | Defines the format and values for positive and negative responses. |
| LC_MONETARY | Affects monetary information returned by the `localeconv()` C function. |

| | |
|---|---|
| LC_NUMERIC | Affects the decimal point character for the formatted input/output and string conversion functions, and the nonmonetary formatting information returned by the `localeconv()` C function. |
| LC_SYNTAX | Defines the variant characters from the Portable Character Set. |
| LC_TIME | Affects the behavior of the `strftime()` C function. |
| LC_TOD | Defines the rules used to define the beginning, end, and duration of daylight savings time, and the difference between local time and Greenwich Mean Time (GMT). |

For related information on creating your own locales, supported categories and locale source file commands, see *z/OS C/C++ Programming Guide*, SC09-4765.

A large number of locales are shipped with the z/OS C/C++ product and are stored in local databases. To refer to a locale, you use a string format. Each locale has a code page associated with it at creation time. This code page is also part of the string name used to refer to the locale.

The generalized format of a string representing a locale is as follows:

```
language[_territory][.code page][@modifier]
```

The following example shows a Danish locale with *language* set to **Da** for Danish, the optional *territory* set to **DK** for Denmark, and the optional *code page* set to **IBM-1047**.

```
Da_DK.IBM-1047
```

Each DCE process or application has an associated current locale. For each category in the current locale, a different locale may be referenced, or a single locale may be referenced for all of the categories.

**Establishing a Current Locale:** The z/OS model follows the POSIX/XPG model of using environment variables and locales. In the POSIX model, the code page is associated with the locale identified by the `LANG` environment variable. Your program adopts the desired code page using the **setlocale(LC_ALL, "")** function call. Refer to *z/OS C/C++ Run-Time Library Reference*, SA22-7821, for a definition of locales. Note that you should establish your locale prior to running any DCE APIs, otherwise your application runs in the default locale for your system. Figure 36 on page 130 shows how locales are established in DCE applications and the conversions performed from the network code page to the application code page at runtime.

If you want to use a locale other than the system's locale in your DCE client or server application, you must establish that locale by using **setlocale()** prior to calling any DCE API. The **setlocale()** function is not supported for multiple POSIX threads. Since DCE uses multiple POSIX threading, you should run **setlocale()** prior to any DCE API in your application. All threads in the process run in the same locale. Since **setlocale()** is process sensitive and not thread sensitive, all threads in the process run in the same locale.

The POSIX definition states that if the **setlocale()** function is not called, the default current locale will be the C or POSIX locale. The default current locale is usually set on a system by the DCE Administrator through environment variables representing each locale category. If you want to change the current locale used in your program, use the ANSI C **setlocale()** function, by declaring it as follows:

```
  char * setlocale(int category, const char * locale);
```

Use the **setlocale()** function to adopt to the user's environment as follows:

```
/*   Set all categories based on the environment variable          */
/*   settings for the process                                      */

setlocale(LC_ALL, "");
```

This sets all the locale categories for your application to the current locale categories for your system. To establish a specific locale for your application, use the following call:

```
setlocale(LC_ALL, locale);
```

For example, to establish the supplied locale for the French language in Canada with code page IBM-037, use the following call:

```
setlocale(LC_ALL, Fr_CA.IBM-037);
```

This establishes `Fr_CA` as the current locale. The code page used with this locale is `IBM-037`. This code page was established and defined in the charmap file that was used to define this supplied locale.

You can also select a locale by using environment variables to specify the names of locale categories. Specifying the environment variables at run time has the same effect as issuing the **setlocale()** function for a category. For example, the following call initializes the environment variable LANG to Da_DK.IBM-1047:

```
setenv("LANG", "Da_DK.IBM-1047", 1);
```

The names of the environment variables that you can set match the names of the locale categories. For more information on setting the locale environment categories using environment variables, see *z/OS C/C++ Run-Time Library Reference*, SA22-7821. For more information on the **LANG** environment variable, see the section that discusses NLS Considerations in *z/OS DCE Messages and Codes*.

**Note:** Once a current locale has been established, you must ensure that all of the character data that is input to your application is in the same code page of that application. If differences in code pages exist, problems may result if variant characters are used in character data. See Figure 38 on page 133.

**Distributed locale Considerations:** With DCE applications, servers may service clients that are running with different locales at the same time. But the server can run in one locale at a time. Depending on how sensitive your server program is to locale dependent items such as sorting, displaying of currency, and so on, you may need to add additional code to your client to fully internationalize your DCE application. To minimize locale sensitive problems, you should run all DCE applications on the local system in the same locale wherever possible.

As previously illustrated, you should not run in multiple code pages on the local system or you may experience a problem handling variant characters. Requests from remote systems are handled for you by normalization of code pages to ISO8859-1 on the wire.

## Code Page Considerations

## Homogeneous Code Page Considerations

If your environment includes multiple z/OS machines that are all running in the same code page, and you want to use a single copy of your application's object code, the situation is simple. You can use the default code page for UNIX System Services, **IBM-1047** or any of the supported code pages listed in Table 7 on page 135, provided you compile your application in the same code page as it will be run, and observe the guidelines for character data presented above.

## Heterogeneous Code Page Considerations

If your environment includes multiple z/OS machines, each running in a different code page, and you want to use one copy of your application's object code, you must fully enable your application for National Language Support.

## DCE-specific Considerations

Following are additional considerations in the areas of:

- Code Page Restrictions
- Client Application Data Considerations
- CDS Clerk.

## Code Page Restrictions

With z/OS DCE code page support, you can utilize your national code page in your DCE applications, but you need to observe the following restrictions:

- On a given UNIX System Services host system running z/OS DCE, you should use one code page for all DCE applications running on that system. The only exception is a DCE application that uses well-known endpoints and does not use DCE CDS or DCE Security data.
- You should only transmit multiple byte character set data such as Kanji *as-is*; that is, declare them as **byte**. Network to local host system code page conversions do not support Kanji or other multiple byte character sets.
- **uuidgen** must be run in code page IBM-1047.
- IDL and ACF files used with z/OS DCE must be coded in code page IBM-1047. If you port your IDL file from another platform, ensure that it is converted to code page IBM-1047. Note that the output files from the z/OS DCE IDL compiler (header files, client and server stub files) will also be in code page IBM-1047.
- The **envar** file, which contains settings for z/OS DCE environment variables, must be maintained in the IBM-1047 code page.
- **dcecp** scripts, which can contain commonly used DCE administration routines, must be in IBM-1047 code page when used. If you port a script file from another platform, be sure that it is converted to code page IBM-1047.
- The **cds_attributes**, **cds_globalnames**, and **xoischema** files, all of which contain information related to CDS, must be maintained in code page IBM-1047.
- When running a control program in interactive mode, the subcommands and arguments entered must be in the local code page. This also applies when running the control program command in batch.

To see the hexadecimal representation of the characters in each code page, refer to Appendix B, "IBM Code Pages" on page 541.

## Client Application Data Considerations

To ensure interoperability with other DCE implementations, client applications should restrict themselves to using character data that belongs to the Portable Character Set (PCS) as defined by POSIX. Only single-byte data should be used, that is; your applications should not use DBCS data when interacting with DCE. Double-byte user data is tolerated within DCE when defined as a transparent byte stream.

All source code for the z/OS C/C++ compiler (including the generated DCE stubs) and user data passed in RPC calls should be stored using the same user-selected code page.

**CDS Clerk:** For z/OS DCE, there is only one CDS clerk to service all of the client application requests. The clients connect with the clerk through local sockets. The CDS clerk daemon process which runs in the system locale is usually started by the DCE Administrator. *If your applications use EBCDIC variant characters, they must run in the same code page as the CDS clerk process on the local system*. If a mismatch occurs, client applications may have problems with EBCDIC variant characters, and this can happen even within the PPCS.

To understand this restriction, refer to Figure 39.

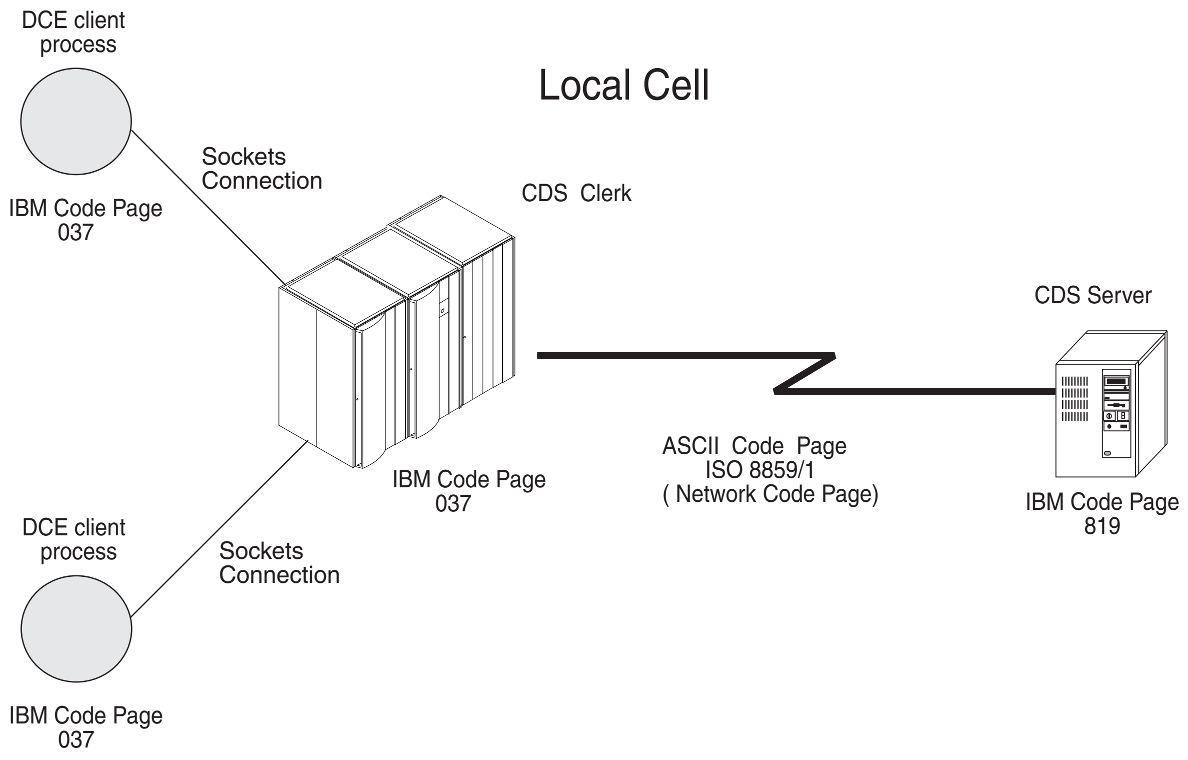


*Figure 39. Restriction on DCE Client Code Set*

When character data is passed between the client and the clerk through the local socket interface, it is passed transparently. No data conversions are performed. Therefore, your client application and the clerk must have the same code page. Because the clerk has the code page of the system associated with it, your client must match this code set. This is a common limitation with interprocess communication on the local system. Note that the DCE Administrator typically sets the CDS Clerk locale. See *z/OS DCE Administration Guide* for the procedure to set the Clerk’s locale.

For information on how ASCII and EBCDIC interoperability is achieved and how CDS data is stored in z/OS DCE, refer to *z/OS DCE Administration Guide*.

# Chapter 9. Writing Internationalized RPC Applications

An "internationalized" DCE RPC application is one that

- Uses the operating system platform's locale definition functions to establish language-specific and culture-specific conventions for the user and programming environment.
- Uses DCE RPC-provided or user-defined character and code set evaluation and automatic conversion features to ensure character and code set interoperability during the transfer of international characters in remote procedure calls between RPC clients and servers.

A “locale” defines the subset of a user's environment that depends upon language and cultural conventions. A locale consists of categories; each category controls specific aspects of some operating system components' behaviors. Categories exist for character classification and case conversion, collation order, date and time formats, numeric non-monetary formatting, monetary formatting, and formats of informative and diagnostic messages and interactive responses. The locale also determines the character sets and code sets used in the environment. The syntax and use of a locale definition function depends on the operating system platform in use with DCE. See the *z/OS C/C++ Programming Guide* for a description of the system's locale definition functions and locale categories.

The remainder of this chapter describes the DCE RPC features for character and code set interoperability in remote procedure calls that are available to programmers who are developing internationalized DCE RPC applications. The first section describes the concepts of character sets, code sets and code set conversion and explains the default character and code set conversion mechanism that the RPC runtime protocol supports for remote procedure calls. The remaining sections describe the execution of a remote procedure call when it uses the DCE RPC features for character and code set interoperability, and explain how to build an RPC application that uses these features.

**Note:** In order to use the internationalization support, z/OS DCE clients and servers must support a code set interface that is at least version 2.0. The code sets interface version is located in **codesets.idl** in the **/usr/lpp/dce/share/include/dce** directory.

## Character Sets, Code Sets, and Code Set Conversion

A “character set” is a group of characters, such as the Latin-1 alphabet, Japanese Kanji, or the European character set. To enable world-wide connectivity, DCE guarantees that a minimum group of characters is supported in DCE. The DCE RPC communications protocol ensures this guarantee by requiring that all DCE RPC clients and servers support the DCE Portable Character Set (PCS). The IDL base type specifiers **char** and **idl_char** identify DCE PCS characters.

A “code set” is a mapping of the members of a character set to specific numeric code values. Examples of code sets include ASCII, JIS X0208 (Japanese Kanji), and ISO 8859-1 (Latin 1). The same character set can be encoded in different code sets; consequently, DCE can contain RPC clients and servers that use the same character set but represent that character set in different numeric encodings. “Code set conversion” is the ability for a DCE RPC client or server to convert character data between different code sets.

The OSF DCE RPC communications protocol, through the Network Data Representation (NDR) transfer syntax, provides automatic code set conversion for DCE PCS characters encoded in two code sets: ASCII and EBCDIC. The RPC communications protocol automatically converts character data declared as **char** or **idl_char** between ASCII and EBCDIC encodings, as necessary, for all DCE RPC clients and servers. Remember that z/OS always converts **char** or **idl_char** data to ISO8859-1 (ASCII) before sending it over the network.

The DCE RPC communications protocol does not provide support for the recognition of characters outside of the DCE PCS, nor does it provide automatic conversion for characters encoded in code sets other than ASCII and EBCDIC.

However, DCE RPC does provide IDL constructs and RPC runtime routines that programmers can use to write RPC applications that exchange non-PCS, or “international” character data that is encoded in code sets other than ASCII and EBCDIC. These features provide mechanisms for international character and code set evaluation and automatic code set conversion between RPC clients and servers. Using these features, programmers can design their applications to run in a DCE environment that supports multiple heterogeneous character sets and code sets.

The next section describes the remote procedure call execution model when the DCE RPC features for character and code set interoperability are used.

## Remote Procedure Call with Character/Code Set Interoperability

Table 2 on page 40 illustrates the basic tasks of an RPC application. Table 8 shows these basic tasks integrated with the additional tasks required to implement an RPC that provides character and code set interoperability.

*Table 8 (Page 1 of 2). Tasks of an Internationalized RPC Application*

| Client Tasks | Server Tasks |
|---|---|
| | 1. Set locale |
| | 2. Select network protocols |
| | 3. Register RPC interfaces |
| | 4. Advertise RPC interfaces and objects in the namespace |
| | 5. Get supported code sets and register them in the namespace |
| | 6. Listen for calls |
| 7. Set locale | |
| 8. Establish character and code sets evaluation routine | |
| 9. Find compatible servers that offer the procedures | |
| 10. **Call the remote procedure** | |
| 11. Establish a binding relationship with the server | |
| 12. Get code set tags from binding handle | |
| 13. Calculate buffer size for possible conversion of input arguments from local to network code set | |
| 14. Convert input arguments from local to network code set (if necessary) | |
| 15. Marshall input arguments | |
| 16. Transmit arguments to the server's runtime | |
| | 17. Receive call |
| | 18. Get code set tags sent from client |
| | 19. Calculate buffer size for possible conversion of input arguments from network to local code set |
| | 20. Unmarshall input arguments |
| | 21. Convert input arguments from network to local code set (if necessary) |
| | 22. Locate and invoke the called procedure |
| | 23. **Execute the remote procedure** |
| | 24. Calculate buffer size for possible conversion of output arguments from local to network code set |

*Table 8 (Page 2 of 2). Tasks of an Internationalized RPC Application*

| Client Tasks | Server Tasks |
|---|---|
| | 25. Convert output arguments from local to network code set (if necessary) |
| | 26. Marshall output arguments and return value |
| | 27. Transmit results to the client's runtime |
| | 28. Remove code set information from namespace on exit |
| 29. Receive results | |
| 30. Calculate buffer size for possible conversion of output arguments from network to local code set | |
| 31. Unmarshall output arguments | |
| 32. Convert output arguments from network to local code set (if necessary) | |
| 33. Pass to the calling code the results and return control to it | |

Following is a list that describes, in more detail, the additional tasks required to implement an “internationalized” RPC application.

1. Both client and server invoke a platform-dependent function to set their locale during initialization. This step establishes the client's and the server's local character and code set; that is, the character and code set currently in use by processes on the client host and processes on the server host.

2. The server, as part of its initialization phase, calls a DCE RPC routine that retrieves information about code sets support on the server's host. The RPC routine examines the host's locale environment and its code set registry to determine the host's “supported code sets”; that is, code sets for which conversion routines exist that processes on the host can use to convert between code sets, if necessary.

   The code set registry is a per-host file that contains mappings between string names for the supported code sets and their unique identifiers. OSF assigns the unique identifiers for the code sets. DCE licensees assign their platform string names for the code sets. The DCE RPC routines for character set and code set interoperability depend upon a code set registry existing on each DCE host. For more information about the code set registry, see the *z/OS DCE Administration Guide*.

   The routine returns a list of the supported code sets to the server; the list consists of each code set's unique identifier.

3. The server next calls a new RPC NSI routine to register the supported code sets information in the name service database. Recall that a server can use the NSI to store its “binding information” (information about its interfaces, objects, and addresses) into its own namespace entry, called a “server entry.” The new RPC NSI routine adds the supported code sets information as an attribute that is associated with the server entry, which the server created when it used the NSI export operation to export its binding information into the name service database. Placing the code sets information into the name service database gives RPC clients access to this information.

4. Before it calls the RPC NSI routines that locate a server that offers the desired remote procedure, the client calls a new RPC routine that sets up a character and code sets compatibility evaluation routine.

5. The client calls RPC NSI routines to locate a compatible server. The RPC NSI routines invoke the character and code sets compatibility evaluation routine set up by the client to evaluate potential compatible servers for character and code set compatibility with the client.

6. The evaluation routine imports the server's supported code sets information from the name service database, retrieves the client's supported code sets information from the client host, and compares the two. If the client and the server are using the same local code set, no code set conversion is necessary, and no data loss will result.

If client and server are using different local code sets, then it is possible that the server is using a different character set than the client. The client does not want to bind to a server that is using a different character set, since significant data loss would result during character data conversion. Consequently, the evaluation routine uses the server's code set information to determine its supported character sets, and rejects servers using incompatible character sets. For example, if a client is using the Japanese Kanji character set (such as JIS0208), the evaluation routine rejects a server that offers the desired remote procedure but which is using the Korean character set (such as KS C 5601).

If the client and server are character set compatible, and they support a common code set into which one or the other (or both) can convert, the evaluation routine deems the server to be compatible with the client. The NSI import routines return this server's binding information to the client.

7. The client makes the remote procedure call.
8. A client stub is called, with the character data represented in the local form and in the local code set.
9. Before marshalling the input arguments, the client stub calls a new stub support routine that retrieves code set identifying information that the evaluation routine established in the binding handle.
10. The client stub next calls a new stub support routine that determines, based on the code set identifying information, whether the character data needs to be converted to another code set, and if so, whether the buffer that currently holds the character data in the local form and code set is large enough to hold the data once it is converted. If the routine determines that conversion is necessary and a new buffer is required, it calculates the size of that buffer and returns the value to the client stub.
11. The client stub next calls a new stub support routine that converts, based on the code set identifying information, the character data from the local code set to the appropriate code set to be used to transmit the data over the network to the server (called the “network code set”).
12. The client stub then marshalls the input arguments and transmits them to the server runtime along with code set identifying information.
13. The server stub is called, with the character data represented in the network form (which is always **idl_byte**) and in the network code set.
14. The server stub calls a new stub support routine that determines, based on the code set identifying information passed in the client call, whether the character data needs to be converted from the network code set to the server's local code set, and if so, whether the buffer that currently holds the character data in the network format and code set is large enough to hold the data once it is converted. If the routine determines that conversion is necessary and a new buffer is required, it calculates the size of that buffer and returns the value to the server stub.
15. The server stub next unmarshalls the input arguments.
16. The server stub next calls a new stub support routine that converts, based on the code set identifying information, the character data from the code set used on the network to the server's local code set.
17. The server stub invokes the manager routine to execute the remote procedure.
18. Before marshalling the results of the remote procedure (the output arguments and return values), the server calls a new stub support routine to determine whether conversion from the server's local code set is necessary, based on the code set identifying information it received from the client, and whether or not the buffer currently holding the character data is large enough to accommodate the converted data. If a new buffer is required, the stub support routine calculates the size of this new buffer and returns it to the server stub.
19. The server stub next calls a new stub support routine that converts, based on the code set identifying information from the client, the character data from the server's local code set to the network code set.
20. The server stub marshalls the converted output arguments and transmits them to the client runtime along with code set identifying information.

21. The server initialization procedure also contains a call to a new RPC routine that removes the code set information from server entry in the name service database if the server exits or is terminated.
22. The client stub is called, with the character data in network format and code set.
23. The client stub calls a new stub support routine that determines, based on the code set identifying information passed by the server, whether the character data needs to be converted from the network code set to the client's local code set, and if so, whether the buffer that currently holds the character data in the network format and code set is large enough to hold the data once it is converted. If the routine determines that conversion is necessary and a new buffer is required, it calculates the size of that buffer and returns the value to the client stub.
24. The client stub next unmarshalls the output arguments.
25. The client stub next calls a new stub support routine that converts, based on the code set identifying information, the character data from the code set used on the network to the client's local code set.
26. The client stub passes the data to the client in the local format and code set.

Note that the stub conversion routines do not implement code set conversion. Instead, they call POSIX-compliant **iconv** code set conversion routines, which are part of the local operating system. As a result, if the platform to which DCE is ported does not provide these POSIX conversion routines, DCE applications that run on this platform cannot use the DCE RPC character and code set interoperability features.

## Building an Application for Character and Code Set Interoperability

An application programmer who wishes to design his or her RPC application for character and code set interoperability performs the following steps:

1. Writes the interface definition file (**.idl**) to include constructs that will enable automatic code set conversion during remote procedure execution
2. Writes an associated attribute configuration file (**.acf**) for the interface that includes ACF attributes that will enable automatic code set conversion during remote procedure execution
3. Writes the stub support routines that client and server stubs use to carry out automatic code set conversion during a remote procedure call. You can omit this step if you use the stub support routines supplied with DCE.
4. Writes the server code and includes the steps to get the server's supported code sets and export them to the name service database, and to remove them from the name service database upon termination or exit.
5. Writes the client code and includes the steps to set up the character and code set evaluation mechanism.
6. Writes the character and code set compatibility evaluation routine. You can omit this step if you use one of the evaluation routines supplied with DCE.

Note that building an RPC application for character and code set interoperability imposes some restrictions on the application. For example, an application that uses the RPC character and code set interoperability features cannot use customized binding handles. See “The cs_char Attribute” on page 298 for more details on internationalized RPC application restrictions.

The next sections describe the steps just outlined in more detail.

## Writing the Interface Definition File

The interface definition file is where the set of remote operations that constitute the interface are defined. The first step in writing an interface definition file that supports automatic code set conversion is to create a special **typedef** that, when used in operation parameters, represents international character data that can be automatically converted, if necessary, before marshalling and unmarshalling at client and server sites. As described in "The byte Type" on page 239, the data representation for a **byte** data type is guaranteed not to change when the data is transmitted by the RPC communications protocol. Consequently, the special international character data type defined in the **.idl** is always declared to be a **byte** type so that the RPC protocol will not automatically treat it as a DCE PCS character and convert it between ASCII and EBCDIC.

The second step in writing an interface definition file that supports automatic code set conversion is to define, for each operation that will transmit the special international character data type, a maximum of three operation parameters that will "tag" the international characters being passed in the operation's input and output parameters with code set identifying information established during the client-server evaluation and binding procedure. These parameters are:

- The sending tag, which indicates the code set the client is using for international characters it transmits over the network. The sending tag has the **in** parameter attribute, and is applied to international character data declared in the operation's input parameters. If the operation does not specify any international character data as input, then it is not necessary to create this parameter.
- The desired receiving tag, which indicates the code set in which the client prefers to receive international character data sent back from the server as output. The desired receiving tag has the **in** parameter attribute. If the operation does not specify any international character output data, then it is not necessary to create this parameter.
- The receiving tag, which indicates the code set the server is using for international characters it transmits over the network. The receiving tag has the **out** parameter attribute, and is applied to international character data declared in the operation's output parameters. If the operation does not specify any international character output data, then it is not necessary to create this parameter.

You must define these code set tag parameters as unsigned long integers or unsigned long integers passed by reference. The receiving tag parameter must be declared as a pointer to the receiving tag unsigned long integer.

When international character data is to be unmarshalled, the client or server stub needs to have received a description of the code set being used before it receives the data. For this reason, the sending tag parameter must occur in an operation's parameter list before any **in** international character data, and the receiving tag parameter must occur in an operation's parameter list before any **out** international character data. The requirement that a tag must be received before the data it relates to is received also means that a customized binding handle cannot include international characters. This is because a binding handle must be the first parameter in a parameter list.

Here is an example **.idl** file for an interface named "cs_test" that uses the special international character type definition and the code set tag parameters for input and output parameters that are fixed arrays of characters from an international character set:

```
[
uuid(b076a320-4d8f-11cd-b453-08000925d3fe),
version(1.0)
]
interface cs_test
{
        const unsigned short SIZE = 100;
        typedef byte net_byte;

        error_status_t cs_fixed_trans (
                [in] handle_t IDL_handle,
                [in] unsigned long stag,
                [in] unsigned long drtag,
                [out] unsigned long *p_rtag,
                [in] net_byte in_string[SIZE],
                [out] net_byte out_string[SIZE]
        );
}
```

## Writing the Attribute Configuration File

The next step in building an RPC application that supports character and code set interoperability is to create an attribute configuration file (**.acf**) to be associated with the **.idl** file. This **.acf** file uses the following attributes:

- The **cs_char** attribute, which associates the local data type that the application code uses to represent international characters in the local code set with the special **typedef** defined in the **.idl** file. This is a required ACF attribute for an RPC application that passes international character data. "The cs_char Attribute" on page 298 provides complete details on how to specify the **cs_char** ACF attribute and the programming restrictions associated with its use.
- The **cs_stag**, **cs_drtag**, and **cs_rtag** attributes, for each operation in the interface that specifies sending tag, desired receiving tag, and/or receiving tag parameters. These ACF attributes declare the tag parameters defined in the corresponding **.idl** file to be special code set tag parameters. Operations defined in the **.idl** that specify international character in input parameters must use the **cs_stag** attribute. Operations defined in the **.idl** that specify international character in output parameters must use the **cs_drtag** and **cs_rtag** attributes. "The cs_stag, cs_drtag, and cs_rtag Attributes" on page 302 provides complete details on how to specify the **cs_stag**, **cs_drtag**, and **cs_rtag** ACF attributes.
- The **cs_tag_rtn** attribute, which specifies the name of a routine that the client and server stubs will call to set an operation's code set tag parameters to specific code set values. The **cs_tag_rtn** attribute is an optional ACF attribute for internationalized RPC applications; application developers can use it to provide code set tag transparency for callers of their application's operations. See "The cs_tag_rtn Attribute" on page 303 for complete details on how to specify the **cs_tag_rtn** attribute. "Writing the Stub Support Routines" on page 150 provides more information on the role of the tag-setting routine.

Here is the companion **.acf** file for the "cs_test" interface defined in "Writing the Interface Definition File" on page 148:

```
[
explicit_handle
]
interface cs_test
{
        include "dce/codesets_stub";

        typedef [cs_char(cs_byte)] net_byte;

        [comm_status, cs_tag_rtn(rpc_cs_get_tags)] cs_fixed_trans
                (
                [cs_stag] stag,
                [cs_drtag] drtag,
                [cs_rtag] p_rtag );
}
```

**Note:** If you are doing a batch compile, modify the **include** statement for the previous **.acf** file to be **`include "dce/codestub";`**.

The ACF for “cs_test” uses the **cs_char** attribute to define **net_byte** as a data type that represents international characters. Note that the local type specified in the **cs_char** attribute definition is **cs_byte**. This local type is analogous to the **byte** type. The ACF for “cs_test” also uses the **cs_tag_rtn** attribute to specify a tag-setting routine.

## Writing the Stub Support Routines

When you use the **cs_char** attribute to define an international character data type, you must provide stub support routines that check the buffer storage requirements for character data to be converted and stub support routines that perform the conversions between the local and network code sets. If you use the **cs_tag_rtn** attribute, you must provide the routine that sets the code set tag parameters for the operations in the application that transfer international characters.

DCE RPC provides several buffer-sizing routines and one tag-setting routine. You can use the DCE RPC routines, or you can develop your own customized buffer-sizing and tag-setting routines; the choice depends upon your application's requirements. The next sections describe these types of stub support routines in more detail.

**Buffer Sizing Routines:** Different code sets use different numbers of bytes to encode a single character. Consequently, there is always the possibility that the converted string can be larger than the original string when converting data from one code set to another. The function of the buffer sizing routines is to first calculate the necessary buffer size for code set conversion between local and network code sets. The buffer sizing routines will then return their findings to the client and server stubs, which call these routines before marshalling and unmarshalling any international character data. The stubs allocate a new buffer, if necessary, before calling the code set conversion routines.

You must provide the following buffer sizing routines for each local type that you define with the **cs_char** attribute:

| | |
|---|---|
| *local_type_name*_**net_size()** | Calculates the necessary buffer size for code set conversion from a local code set to a network code set. Client and server stubs call this routine before they marshall any international character data. |
| *local_type_name*_**local_size()** | Calculates the necessary buffer size for code set conversion from a network code set to a local code set. Client and server stubs call this routine before they unmarshall any international character data. |

You specify the name for the local data type in the *local_type_name* portion of the function name and the appropriate suffix name (**_net_size** or **_local_size**).

DCE RPC provides buffer sizing routines for the **cs_byte** data type. The **cs_byte** data type is equivalent to the **byte** type.

The DCE RPC buffer sizing routines are:

| | |
|---|---|
| **cs_byte_net_size()** | Calculates the necessary buffer size for code set conversion from a local code set to a network code set when the **cs_byte** type has been specified as the local data type in the **.acf** file. |
| **cs_byte_local_size()** | Calculates the necessary buffer size for code set conversion from a network code set to a local code set when the **cs_byte** type has been specified as the local data type in the **.acf** file. |

If your internationalized RPC application uses **cs_byte** as the local type in the ACF, it can use these DCE RPC buffer sizing routines; in order to do so, simply link with the DCE library when compiling your application. The example ACF shown earlier in this chapter uses the **cs_byte** type as the local type. Consequently, the client and server stubs will use the **cs_byte_** buffer sizing routines. Refer to the *z/OS*

*DCE Application Development Reference* for a description of the **cs_byte_** routine signatures and functions.

Applications that use data types other than **cs_byte** as their local data types will need to provide their own buffer sizing routines. User-provided buffer sizing routines must follow the same signature as the DCE RPC-provided buffer sizing routines. See the *z/OS DCE Application Development Reference* for a description of the **cs_byte_** required routine signatures.

**Note:** Do not send wide character data over the network using the **cs_byte** data type. Always convert the data back to its multi-byte format before making the RPC call.

**Code Set Conversion Routines:** When RPC clients and servers exchange international character data, the data being exchanged needs to be understood by both client and server. Both client and server need to understand a character set, and both client and server need to understand the way that character set is expressed. Code set conversion provides a way for a character set to be represented in a form that both client and server can understand, given that the client and server are using a compatible character set. (In general, character set conversion is not recommended; it is unlikely that clients and servers would want to map, for example, German characters to Chinese characters due to the data loss that would occur as a result.)

The stub support routines for code set conversion provide the mechanism for the stubs to use to convert between different code sets, given that character set compatibility has been established. The code set conversion routines translate a character set from one encoding to another. Consequently, the code set conversion routines provide the way for a character set to be represented in a form that both client and server can understand.

You must provide the following code set conversion routines for each local type that you define with the **cs_char** attribute:

| | |
|---|---|
| *local_type_name*_**to_netcs()** | Converts international character data from a local code set to a network code set. Client and server stubs call this routine before they marshall any international character data. |
| *local_type_name*_**from_netcs()** | Converts international character data from a network code set to a local code set. Client and server stubs call this routine after they unmarshall any international character data. |

You specify the name for the local data type in the *local_type_name* portion of the function name and the appropriate suffix name **_to_netcs** or **_from_netcs**.

DCE RPC provides code set conversion routines for the **cs_byte** data type. These routines are:

| | |
|---|---|
| **cs_byte_to_netcs()** | Converts international character data from a local code set to a network code set when the **cs_byte** type has been specified as the local data type in the **.acf** file. |
| **cs_byte_from_netcs()** | Converts international character data from a network code set to a local code set when the **cs_byte** type has been specified as the local data type in the **.acf** file. |

If your application uses the **cs_byte** data type as the local type, it can use these DCE RPC code set conversion routines; in order to do so, simply link with the DCE library when compiling your application. Refer to the *z/OS DCE Application Development Reference* for a description of the **cs_byte_** routine signatures and functions.

Applications that use data types other than **cs_byte** as their local data types will need to provide their own code set conversion routines. User-provided code set conversion routines must follow the same signature as the DCE RPC-provided code set conversion routines. See the *z/OS DCE Application Development Reference* for a description of the **cs_byte_** routine signatures and functions.

**Note:** Do not send wide character data over the network using the **cs_byte** data type. Always convert the data back to its multi-byte format before making the RPC call.

The DCE code set conversion routines depend upon the presence of the XPG4 **iconv** code set conversion facility in the underlying operating system platform. The **iconv** facility consists of the following routines:

| | |
|---|---|
| **iconv_open()** | Code conversion allocation function; returns a conversion descriptor that describes a conversion from the code set specified in one string pointer argument to the code set specified in another string pointer argument. |
| **iconv()** | Code conversion function; converts the sequence of characters from one code set into a sequence of corresponding characters in another code set. |
| **iconv_close()** | Code conversion deallocation function; deallocates the conversion descriptor and all associated resources allocated by the **iconv_open()** function. |

Note that the **iconv** facility identifies a code set by a string name. This string name is the name that the local platform uses to refer to the code set. However, all of the stub support routines for automatic code set conversion use the unique identifier assigned to the code set in the code set registry to identify a code set. Before the DCE code set conversion routines can invoke the **iconv** facility, they must access the code set registry to retrieve the platform-specific string names associated with the local and network code set identifiers.

The DCE code set conversion routines use the **dce_cs_loc_to_rgy()** and **dce_cs_rgy_to_loc()** routines to access the code set registry and translate between code set string names and their corresponding unique identifiers. The *z/OS DCE Application Development Reference* provides a description of these routines' signatures and functions; developers who are writing their own code set conversion routines and who are using the **iconv** facility for conversion may want to use these DCE routines to convert between code set names and identifiers. See the *z/OS DCE Administration Guide* for the code set registry shipped with z/OS.

**Tag-Setting Routine:** Recall from “Writing the Interface Definition File” on page 148 that operations that specify international characters as input and output parameters declare special code set tag parameters. The purpose of these parameters is to hold the unique identifier for the code set into which the input or output data is to be encoded when it is transferred over the network.

The function of the tag-setting routine is to provide a way to set an operation's code set tag parameters to specific code set values from within the stubs rather than in the application code. The application specifies the name of the tag-setting routine as the argument to the **cs_tag_rtn** ACF attribute. The client and server stubs call this routine to set the tag parameters to specific network code set values before they call the stub support routines for buffer sizing and code set conversion. The stubs use the network code set values returned by the tag-setting routine as input to the buffer sizing and conversion routines. In turn, these routines compare the network code set values to be used for input and output data to the local code set in use for the data, and determine whether or not new buffer allocation and code set conversion are necessary.

When called from the client stub, the tag-setting routine sets the sending tag parameter to the code set to use for input character data. If the client expects output character data from the server, the routine also sets the desired receiving tag parameter to the code set that the client prefers the server to use for sending back the output data. On the client side, the *local_type_name*_**net_size()** buffer sizing routine and the *local_type_name*_**to_netcs()** code set conversion routines use the value in the sending tag as the network code set value to use for transmitting the input data. When the input data arrives at the server side, the server stub uses the sending tag as input to the *local_type_name*_**local_size()** buffer sizing routine and the *local_type_name*_**from_netcs()** code set conversion routines. These routines use the value to determine whether new buffer allocation and conversion is necessary from the network code set to the local code set.

When called from the server stub, the tag-setting routine sets the receiving tag parameter to the code set to use for transmitting the output character data back to the server. The routine can use the desired receiving tag value as input to determine the most appropriate code set in which to encode output data for the client. On the server side, the *local_type_name*_**net_size()** buffer sizing routine and the *local_type_name*_**to_netcs()** code set conversion routines use the value in the receiving tag as the network code set value to use for transmitting the output data. When the output data arrives at the client side, the client stub uses the receiving tag as input to the *local_type_name*_**local_size()** buffer sizing routine and the *local_type_name*_**from_netcs()** code set conversion routines. These routines use the value to determine whether or not new buffer allocation and conversion is necessary from the network code set to the local code set.

DCE RPC provides one tag-setting routine named **rpc_cs_get_tags()** that applications can use to set code set tag values within the stubs. To use this routine, specify its name as the argument to the **cs_tag_rtn** attribute and link your application with the DCE library. The example ACF for the “cs_test” interface specifies the **rpc_cs_get_tags()** routine.

Note that the **rpc_cs_get_tags()** routine always sets the receiving tag value on the server side to the value that the client specified in the desired receiving tag. See the *z/OS DCE Application Development Reference* for an explanation of this routine's signature and function.

RPC application programmers who are developing their own tag-setting routines can also refer to the *z/OS DCE Application Development Reference* to obtain the required signature for their user-written routine.

The tag-setting routine generally obtains the code set tag values from the binding handle. These values are usually determined by the character and code sets evaluation routine invoked during the server binding import process, although they can be explicitly set in the binding handle by using the **rpc_cs_binding_set_tags()** routine. However, applications can design the tag-setting routine to perform evaluation within the stubs rather than in the application (client) code. For example, when called from the client side, the DCE RPC tag-setting routine **rpc_cs_get_tags()** performs character and code set compatibility evaluation itself if it does not find the tag values in the binding handle. See “Writing the Evaluation Routine” on page 163 for more information on deferred evaluation.

## Writing the Server Code

A programmer who is developing an RPC server that supports character and code set interoperability needs to add the following steps to the server's initialization functions in addition to the normal initialization functions it carries out for RPC:

- Setting the server's locale
- Establishing the server's supported code sets
- Registering the server's supported code sets in the name service database
- Establishing a cleanup function that removes the server's supported code sets from the name service database on the server's termination or exit.

The next sections explain these steps in detail.

**Setting the Server's Locale:** The server initialization code needs to include a platform-specific routine that sets the locale environment for the server. This step establishes

- The name of the server's “local code set,” which is the code set that processes on the host will use to encode character data.
- The names of the code sets for which converters exist on the host and consequently, into which processes that run on the host can convert if necessary.

An example of a locale-setting function is the POSIX, XPG3, XPG4 **setlocale()** function, which is defined in **locale.h**. The server code should call the locale-setting function as the first step in the initialization code, before calling the DCE RPC routines that register the interface and export the binding information.

The locale-setting function also establishes the value for two platform-specific macros that indicate:

- The maximum number of bytes the local code set uses to encode one character.
- The maximum number of bytes that any of the supported code sets on the host will use to encode one character.

On POSIX, XPG3, and XPG4 platforms, these macros are **MB_CUR_MAX** and **MB_LEN_MAX** and are defined in **stdlib.h** and **limits.h**, respectively. The buffer sizing routines use **MB_CUR_MAX** when calculating the size of a new buffer to hold converted character data.

Note that all hosts that are members of an “internationalized” DCE cell, that is, a cell that supports internationalized RPC applications, must provide converters that convert between their supported code sets and the ISO 10646 “universal” code set. The DCE RPC functions for character and code set interoperability use the universal code set as the default “intermediate” code set into which a client or server can convert if there are no other compatible code sets between them. “Writing the Evaluation Routine” on page 163 discusses code set evaluation in more detail.

**Establishing the Server's Supported Code Sets:** The next step in writing an internationalized RPC server is to add to the server's initialization code a call to the DCE RPC routine **rpc_rgy_get_codesets()**. This routine gets the supported code set names defined in the locale environment and translates those names to their unique identifiers by accessing the code set registry on the host. The server initialization code should call this routine after it has registered the interface and created a server entry for its binding information in the name service database (by calling the DCE RPC NSI binding export routine **rpc_ns_binding_export()**).

The routine returns an array of unique identifiers from the code set registry that correspond to the server's local code set and the code sets into which the server can convert, if necessary; this data structure is called the “code sets array.” The code sets array also contains, for each code set, the maximum number of bytes that code set uses to encode one character. For an example of the z/OS code set registry, see the *z/OS DCE Administration Guide*.

The purpose of this step is to obtain the registered unique identifiers for the server's supported code sets for use by the DCE character and code set interoperability features, rather than using the string names for the code sets. The DCE features for character and code set interoperability do not use string names because different operating systems commonly use different string names to refer to the same code set. Clients and servers passing international characters in a cell of heterogeneous platforms need to ensure that they both refer to the same code set when establishing compatibility. The code set registry provides the means for clients and servers to uniquely identify a code set while permitting different platforms and the code set converters offered on those platforms to continue to use the string names for the code sets.

See the *z/OS DCE Application Development Reference* for a description of the **rpc_rgy_get_codesets()** routine's signature and arguments.

**Registering the Server's Supported Code Sets in the Namespace:** The next step in writing an internationalized RPC server is to make a call in the server's initialization code to the DCE RPC routine **rpc_ns_mgmt_set_attribute()**, which takes the code sets array returned by **rpc_rgy_get_codesets()** and exports it to the server's entry in the name service database. The routine creates a "code sets" NSI attribute in the name service database and associates it with the server entry created by the NSI export operation.

The purpose of this step is to register the server's supported code sets into the name service database so that clients can gain access to the information. Note, then, that server entries for internationalized RPC servers will have code sets attributes in addition to the binding attributes and object attributes for the servers. For a general discussion of NSI attributes, see "Directory Services and RPC: Using the Namespace" on page 96. Refer to the *z/OS DCE Application Development Reference* for a description of the **rpc_ns_mgmt_set_attribute()** routine's signature and arguments.

**Establishing a Cleanup Function for the Namespace:** The last step in writing an internationalized RPC server is to add a call to the DCE RPC routine **rpc_ns_mgmt_remove_attribute()** to the cleanup code within the server's initialization code. This DCE RPC routine will remove the code sets attribute associated with the server entry from the name service data base when it is called from the cleanup code as the result of a server crash or exit. See the *z/OS DCE Application Development Reference* for a description of the **rpc_ns_mgmt_remove_attribute()** routine's signature and arguments.

**Sample Server Code:** Here is an example of an internationalized RPC server that exports the "cs_test" interface defined in "Writing the Interface Definition File" on page 148.

```
#include <stdio.h>
#include <stdlib.h>
#include <dce/rpc.h>
#include <dce/nsattrid.h>
#include <dce/dce_error.h>
#include <locale.h>
#include <pthread.h>
#include <dce/codesets.h>
#include "cs_test.h"

/*
 * Macro for result checking
 */

#define CHECK_STATUS(t, func, returned_st, expected_st) \
{ \
        if (returned_st == expected_st) { \
        } \
        else { \
                dce_error_inq_text(returned_st, \
                 (unsigned char *)unexpected, &dce_status); \
                dce_error_inq_text(expected_st,\
                 (unsigned char *)expected, &dce_status); \
                printf("FAILED %s()\nresult:   %s\nexpected:  %s\n\n", \
                        func, unexpected, expected); \
                if (t) \
                       return; \
                } \
        } \
} \

static unsigned char   unexpected[dce_c_error_string_len];
static unsigned char   expected[dce_c_error_string_len];
static int             dce_status;
```

```
int
main(int argc, char *argv[])
{
        error_status_t          status;
        int                     i;
        rpc_ns_handle_t         inq_contxt;
        rpc_binding_vector_t    *binding_vector;
        rpc_codeset_mgmt_p_t    arr;
        pthread_t               this_thread = pthread_self();
        sigset_t                sigset;
        char                    nsi_entry_name[256];
        char                    *get_nsi_entry_name;
        char                    *server_locale_name;
        error_status_t          expected = rpc_s_ok;
        int                     server_pid;

        /* The environment variable I18N_SERVER_ENTRY needs
         * to be set before running this program. This is
         * not a DCE environment variable, so you can set up
         * your own environment variable if you like.
         */

             get_nsi_entry_name = getenv("I18N_SERVER_ENTRY");
             strcpy(nsi_entry_name, get_nsi_entry_name);

             (void)pthread_mutex_init(&mutex, pthread_mutexattr_default);

             /* Set the locale. In this way, the current locale
              * information is extracted from XPG/POSIX defined
              * environment variable LANG or LC_ALL.
              */

             setlocale(LC_ALL, "");

             /*
              * Get supported code sets.
              */
             rpc_rgy_get_codesets (
                     &arr,
                     &status );

             CHECK_STATUS(TRUE, "rpc_rgy_get_codesets", status, expected);


             rpc_server_register_if (
                     cs_test_v1_0_s_ifspec,
                     NULL,
                     NULL,
                     &status );

             CHECK_STATUS(TRUE, "rpc_server_register_if", status, expected);


             rpc_server_use_all_protseqs (
                     rpc_c_protseq_max_reqs_default,
                     &status );

             CHECK_STATUS(TRUE, "rpc_server_use_all_protseqs", status, expected);


             rpc_server_inq_bindings (
                     &binding_vector,
                     &status );
```

```
CHECK_STATUS(TRUE, "rpc_server_inq_bindings", status, expected);


rpc_ep_register (
        cs_test_v1_0_s_ifspec,
        binding_vector,
        NULL,
        NULL,
        &status );

CHECK_STATUS(TRUE, "rpc_ep_register", status, expected);


rpc_ns_binding_export (
        rpc_c_ns_syntax_default,
        (unsigned_char_p_t)nsi_entry_name,
        cs_test_v1_0_s_ifspec,
        binding_vector,
        NULL,
        &status );

CHECK_STATUS(TRUE, "rpc_ns_binding_export", status, expected);


/*
 * Register the server's supported code sets into the name space.
 */

rpc_ns_mgmt_set_attribute (
        rpc_c_ns_syntax_default,
        (unsigned_char_p_t)nsi_entry_name,
        rpc_c_attr_codesets,
        (void *)arr,
        &status );

CHECK_STATUS(TRUE, "rpc_ns_mgmt_set_attribute", status, expected);

/*
 * Free memory allocated by getting code sets.
 */
rpc_ns_mgmt_free_codesets (&arr, &status);

CHECK_STATUS(TRUE, "rpc_ns_mgmt_free_codeset", status, expected);

sigemptyset(&sigset);
sigaddset(&sigset, SIGINT);

if (pthread_signal_to_cancel_np(&sigset, &this_thread) != 0)
{
        printf("pthread_signal_to_cancel_np failed\n");
        exit(1);
}
TRY
{
        server_pid = getpid();

        printf("Listening for remote procedure calls...\n");

        rpc_server_listen (
                rpc_c_listen_max_calls_default,
                &status );

                CHECK_STATUS(TRUE, "rpc_server_listen", status, expected);
        /*
         * Remove code set attributes from namespace on return.
         */

        rpc_ns_mgmt_remove_attribute (
                rpc_c_ns_syntax_default,
                (unsigned_char_p_t)nsi_entry_name,
                rpc_c_attr_codesets,
                &status );

        CHECK_STATUS(TRUE, "rpc_ns_mgmt_remove_attribute", status, expected);

        rpc_ns_binding_unexport (
                rpc_c_ns_syntax_default,
                (unsigned_char_p_t)nsi_entry_name,
                cs_test_v1_0_s_ifspec,
                (uuid_vector_p_t)NULL,
                &status );

        CHECK_STATUS(TRUE, "rpc_ns_binding_unexport", status, expected);

        rpc_ep_unregister (
                cs_test_v1_0_s_ifspec,
                binding_vector,
                NULL,
                &status );

        CHECK_STATUS(TRUE, "rpc_ep_unregister", status, expected);

        rpc_binding_vector_free (
                &binding_vector,
                &status );

        CHECK_STATUS(TRUE, "rpc_binding_vector_free", status, expected);

        rpc_server_unregister_if (
                cs_test_v1_0_s_ifspec,
                NULL,
                &status );

        CHECK_STATUS(TRUE, "rpc_server_unregister_if", status, expected);

        (void)pthread_mutex_destroy(&mutex);
}
CATCH_ALL
{
        /*
         * Remove code set attribute from namespace on a signal.
         */

        rpc_ns_mgmt_remove_attribute (
                rpc_c_ns_syntax_default,
                (unsigned_char_p_t)nsi_entry_name,
                rpc_c_attr_codesets,
                &status );

        CHECK_STATUS(TRUE, "rpc_ns_mgmt_remove_attribute", status, expected);

        rpc_ns_binding_unexport (
                rpc_c_ns_syntax_default,
                (unsigned_char_p_t)nsi_entry_name,
                cs_test_v1_0_s_ifspec,
                (uuid_vector_p_t)NULL,
                        &status );

                CHECK_STATUS(TRUE, "rpc_ns_binding_unexport", status, expected);

                rpc_ep_unregister (
                        cs_test_v1_0_s_ifspec,
                        binding_vector,
                        NULL,
                        &status );

                CHECK_STATUS(TRUE, "rpc_ep_unregister", status, expected);

                rpc_binding_vector_free (
                        &binding_vector,
                        &status );

                CHECK_STATUS(TRUE, "rpc_binding_vector_free", status, expected);


                rpc_server_unregister_if (
                        cs_test_v1_0_s_ifspec,
                        NULL,
                        &status );

                CHECK_STATUS(TRUE, "rpc_server_unregister_if", status, expected);

                (void)pthread_mutex_destroy(&mutex);
        }
        ENDTRY;
}
```

## Writing the Client Code

A programmer who is developing an RPC client that supports character and code set interoperability needs to add the following steps to the client code in addition to the basic functions for RPC:

- Setting the client's locale
- Establishing a character and code sets compatibility evaluation routine that the NSI server binding import routines will call to evaluate potential servers for character and code set compatibility

The next sections explain these steps in detail.

**Setting the Client's Locale:** The first step in developing an internationalized RPC client is to add a call within the client code to a platform-specific function that sets the locale environment for the client. This step establishes:

- The name of the client's “local code set,” which is the code set that processes on the host will use to encode character data.
- The names of the code sets for which converters exist on the host and consequently, into which processes that run on the host can convert if necessary.

The call to the locale-setting function must be the first call made within the client code. An example of a locale-setting function is the POSIX, XPG3, XPG4 **setlocale()** function, which is defined in **locale.h**.

The locale-setting function also establishes the value for two platform-specific macros that indicate:

- The maximum number of bytes the local code set uses to encode one character.
- The maximum number of bytes that any of the supported code sets on the host will use to encode one character.

On the POSIX, XPG3, XPG4 platform, these macros are **MB_CUR_MAX** and **MB_LEN_MAX** and are defined in **stdlib.h** and **limits.h**, respectively. The buffer sizing routines use the **MB_CUR_MAX** macro when calculating the size of a new buffer to hold converted character data.

Note that all hosts that are members of an internationalized DCE cell must provide converters that convert between their supported code sets and the ISO 10646 “universal” code set. The DCE RPC functions for character and code set interoperability use the universal code set as the default “intermediate” code set into which a client or server can convert if there are no other compatible code sets between them. “Writing the Evaluation Routine” on page 163 discusses code set evaluation in more detail.

**Establishing the Compatibility Evaluation Routine:** The last step in writing an internationalized RPC client is to call the DCE RPC NSI routine **rpc_ns_import_ctx_add_eval()**. The purpose of this NSI routine is to add evaluation routines to the import context created by the **rpc_ns_binding_import_begin()** routine that the NSI routine **rpc_ns_binding_import_next()** will call to perform additional compatibility evaluation on potential servers.

The internationalized RPC client code calls the **rpc_ns_import_ctx_add_eval()** routine to set up one or more character and code sets compatibility evaluation routines to be called from **rpc_ns_binding_import_next()**. The client code must make the call to **rpc_ns_import_ctx_add_eval()** once for each compatibility routine that it wants to add to the import context for **rpc_ns_binding_import_next()**. See the *z/OS DCE Application Development Reference* for a description of the **rpc_ns_import_ctx_add_eval()** routine's signature and arguments.

The **rpc_ns_import_ctx_add_eval()** must be used in conjunction with the **rpc_ns_binding_import_begin/next/done()** suite of RPC NSI binding functions, because these functions provide an import context argument. If you want to use the **rpc_ns_binding_lookup_begin/next/done/select()** suite of RPC NSI routines, your client code will need to perform character and code set evaluation logic on the binding handle returned by **rpc_ns_binding_select()**. “Example Character and Code Sets Evaluation Logic” on page 166 provides a sample client that performs character and code set evaluation in conjunction with the “lookup” and “select” RPC NSI routines.

**Sample Client Code:** Here is an example of an internationalized RPC client that calls the operation defined in the “cs_test” interface shown in “Writing the Interface Definition File” on page 148. The client establishes the DCE RPC evaluation routine **rpc_cs_eval_without_universal()** as the character and code sets evaluation routine to use.

```
#include <stdio.h>
#include <locale.h>
#include <dce/rpc.h>
#include <dce/rpcsts.h>
#include <dce/dce_error.h>

#include "cs_test.h"          /* IDL generated include file */

/*
 * Result check MACRO
 */
#define CHECK_STATUS(t, func, returned_st, expected_st) \
{ \
        if (returned_st == expected_st) { \
/*
 * Do nothing.
 */
        } else { \
                dce_error_inq_text(returned_st,\
                 (unsigned char *)unexpected, &dce_status); \
                dce_error_inq_text(expected_st, \
                (unsigned char *)expected, &dce_status); \
                printf("FAILED %s()\nresult:   %s\nexpected: %s\n\n", \
                    func, unexpected, expected); \
                if (t) \
                        return; \
                } \
        } \
} \

static unsigned char         unexpected[dce_c_error_string_len];
static unsigned char         expected[dce_c_error_string_len];
static int                   dce_status;

void
main(void)
{
        rpc_binding_handle_t    bind_handle;
        rpc_ns_handle_t         import_context;
        error_status_t          status;
        error_status_t          temp_status;
        cs_byte                 net_string[SIZE];
        cs_byte                 loc_string[SIZE];
        unsigned char           err_buf[256];
        char                    nsi_entry_name[256];
        char                    *get_nsi_entry_name;
        char                    *client_locale_name;
        int                     i, rpc_num;
        FILE                    *fp_in, *fp_out;

        /* The environment variable I18N_SERVER_ENTRY needs
         * to be set before running this program. This is
         * not a DCE environment variable, so you can set up
         * your own environment variable if you like.
         */

        get_nsi_entry_name = getenv("I18N_SERVER_ENTRY");
        strcpy(nsi_entry_name, get_nsi_entry_name);

        setlocale(LC_ALL, "");

        rpc_ns_binding_import_begin (
                rpc_c_ns_syntax_default,
                (unsigned_char_p_t)nsi_entry_name,
                cs_test_v1_0_c_ifspec,
                NULL,
                &import_context,
                &status );

        CHECK_STATUS(TRUE, "rpc_ns_binding_import_begin", status, rpc_s_ok);

        /*
         *  Add code set compatibility checking logic to the context.
         */
        rpc_ns_import_ctx_add_eval (
                &import_context,
                rpc_c_eval_type_codesets,
                (void *)nsi_entry_name,
                rpc_cs_eval_without_universal,
                NULL,
                    &status );

        CHECK_STATUS(TRUE, "rpc_ns_import_ctx_add_eval", status, rpc_s_ok);
```

```
while (1) {
        rpc_ns_binding_import_next (
                  import_context,
                  &bind_handle,
                  &status );

        CHECK_STATUS(FALSE, "rpc_ns_binding_import_next", status, rpc_s_ok);
        if (status == rpc_s_ok)
                  break;
        else
        {
                  return;
        }
}

rpc_ns_binding_import_done (
        &import_context,
        &status );

CHECK_STATUS(TRUE, "rpc_ns_binding_import_done", status, rpc_s_ok);

rpc_ep_resolve_binding (bind_handle,
        cs_test_v1_0_c_ifspec,
        &temp_status);

CHECK_STATUS(TRUE, "rpc_ep_resolve_binding", temp_status, rpc_s_ok);

if(rpc_mgmt_is_server_listening(bind_handle, &status)
        && temp_status == rpc_s_ok)
{
        ; /* Do nothing. */
}
else
{
        dce_error_inq_text ((unsigned long)status,
                err_buf, (int *)&temp_status);
        printf("is_server_listening error -> %s\n", err_buf);
}

/*
 *  This program reads the data from a file.
 */

fp_in = fopen("./i18n_input_data", "r");

if (fp_in == NULL)
{
        printf("i18n_input_data open failed\n");
        return;
}

fp_out = fopen("./i18n_method_fixed_result_file", "w");

if (fp_out == NULL)
{
        printf("i18n_result_file open failed\n");
        fclose(fp_in);
        return;
}

rpc_num = 1;
(void)fgets((char *)net_string, SIZE, fp_in);

while (!feof(fp_in))
```

```
        {

                temp_status = cs_fixed_trans(bind_handle, net_string, loc_string);

                if (temp_status != rpc_s_ok)
                {
                        dce_error_inq_text(temp_status, err_buf,
                        (int *)&status);

                        printf("FAILED %ld  MSG: %s\n", (unsigned long)temp_status, err_buf);
                }
                else
                {
                        printf("PASSED rpc #%d\n", rpc_num++);
                        (void)fputs((char *)loc_string, fp_out);
                        (void)fputs("\n", fp_out);
                }
                (void)fgets((char *)net_string, SIZE, fp_in);
        }

          fclose(fp_in);
          fclose(fp_out);

          return;
}
```

## Writing the Evaluation Routine

Recall from the *z/OS DCE Application Development Guide: Introduction and Style* and Chapter 4, "Developing a Simple RPC Application" on page 39 that when a prospective client attempts to import binding information from a namespace entry that it looks up by name, the NSI import routine checks the binding for compatibility with the client by comparing interface UUIDs and protocol sequences. If the UUIDs match and the protocol sequences are compatible, the NSI operation considers the binding handle contained in the server entry to be compatible and returns it to the client. Internationalized clients need to perform additional compatibility checking on potential server bindings: they need to evaluate the server for character and code set compatibility.

The purpose of the character and code sets compatibility evaluation routine is to determine:

- Whether the character set the server supports is compatible with the client's character set, since incompatible character sets result in unacceptable data loss during character conversion.
- The level of code sets compatibility between client and server, which determines the "conversion method" that the client and server will use when transferring character data between them.

A conversion method is a process for converting one code set into another. There are four conversion methods:

- Receiver Makes It Right (RMIR) — the recipient of the data is responsible for converting the data from the sender's code set to its own code set. This is the method that the RPC communications protocol uses to convert PCS character data between ASCII and EBCDIC code sets.
- Client Makes It Right (CMIR) — the client converts the input character data to be sent to the server into the server's code set before the data is transmitted over the network, and converts output data received from the server from the server's code set into its local code set.
- Server Makes It Right (SMIR) — the server converts the input character data received from the client into its local code set from the client's code set and converts output data to be sent to the client into the client's code set before the data is transmitted over the network.

- Intermediate — Both client and server convert to a common code set. DCE defines a default intermediate code set to be used when there is no match between the client and server's supported code sets; this code set is the ISO 10646 “universal” code set.

A character and code sets compatibility evaluation routine generally employs a “conversion model” when determining the level of code sets compatibility. A conversion model is an ordering of conversion methods, for example, “CMIR first, then SMIR, then intermediate.” Consequently, the actual conversion method used is determined at runtime.

**DCE RPC Evaluation Routines:** DCE RPC provides two character and code sets compatibility evaluation routines: **rpc_cs_eval_with_universal()** and **rpc_cs_eval_without_universal()**. To use either one of these routines, specify their names in the evaluation function argument to the **rpc_ns_import_ctx_add_eval()** routine. (The sample client code shown in “Sample Client Code” on page 160 specifies a DCE RPC character and code sets evaluation routine.)

The **rpc_cs_eval_with_universal()** routine first compares the client's local code set with the server's local code set. If they are the same, client-server character and code set compatibility exists. The routine returns to the NSI import routine, which returns the server binding to the client.

If the routine finds that the client and server local code sets differ, it calls the routine **rpc_cs_char_set_compat_check()** to determine client-server character set compatibility. If the client and server are using the same character set, it will be safe for them to exchange character data despite their use of different encodings for the character data. Clients and servers using different character sets are considered to be incompatible, since the process of converting the character data from one character set to the other will result in significant data loss.

Using the client and server's local code set identifiers as indexes into the code set registry, the **rpc_cs_char_set_compat_check()** routine obtains the registered values that represent the character set(s) that the specified code sets support. If the client and server support just one character set, the routine compares the values for compatibility. If the values do not match, then the client-server character sets are not compatible; for example, the client is using the German character set and the server is using the Korean character set. In this case, the routine returns the status code **rpc_s_ss_no_compat_charsets** to the evaluation routine so that binding to that server will be rejected.

If the client and server support multiple character sets, the **rpc_cs_char_set_compat_check()** routine determines whether at least two of the sets are compatible. If two or more sets match, the routine considers the character sets compatible, and returns a success status code to the evaluation routine.

In the case where the client and server are character set compatible, the **rpc_cs_eval_with_universal()** routine uses the following model to determine a conversion method:

- RMIR (receiver makes it right)
- SMIR (client uses its local, server converts to and from it)
- CMIR (server uses its local, client converts to and from it)
- Use the universal (ISO 10646) code set as the intermediate code set

This conversion model translates into the following steps:

- The **rpc_cs_eval_with_universal()** routine takes the client's local code set and searches through the server's code sets array to determine whether it has a converter for the client's local set. Then it takes the server's local code set and searches through the client's code sets array to see if it has a converter for the server's local code set.
- If both client and server support converters for each others' local code sets, that is, they can convert to and from each other's local code set, the routine sets the conversion method to RMIR.

- If the server can convert to and from the client's local code set, but the client cannot convert from the server's local code set, the routine sets the conversion method to SMIR.
- If the client can convert to and from the server's local code set, but the server cannot convert to and from the client's local code set, the routine sets the conversion method to CMIR.

  If the conversion method is SMIR or RMIR, the **rpc_cs_eval_with_universal()** routine sets both the sending tag and the desired receiving tag to the code set value that represents the client's local code set. In the case of CMIR, the routine sets both the sending tag and the desired receiving tag to the code set value that represents the server's local code set.
- If neither the client nor server support each other's local code set, the routine sets the sending tag and desired receiving tag to the code set value that represents the ISO 10646 universal code set, which is the default intermediate code set that all DCE clients and servers support.

The **rpc_cs_eval_without_universal()** routine uses the following conversion model to determine a conversion method:

- RMIR
- SMIR (client uses its local, server converts to and from it)
- CMIR (server uses its local, client converts to and from it)
- Reject for code set incompatibility

Consequently, the **rpc_cs_eval_without_universal()** uses the same evaluation logic as **rpc_cs_eval_with_universal()** except that it rejects the server binding if the client and server do not support each other's local code set.

**Writing Customized Evaluation Routines:** Programmers writing internationalized RPC applications can develop their own character and code sets compatibility evaluation routines if their applications' needs are not met by the DCE RPC evaluation routines. These programmers may want to use the following DCE RPC routines within their evaluation routine:

- The **rpc_rgy_get_codesets()** routine
- The **rpc_cs_char_set_compat_check()** routine
- The **rpc_cs_binding_set_tags()** routine
- The **dce_cs_loc_to_rgy()** routine
- The **rpc_ns_mgmt_read_codesets()** routine
- The **rpc_ns_mgmt_free_codesets()** routine

Refer to the *z/OS DCE Application Development Reference* for complete details about these routines.

Programmers who write their own evaluation routines can also select when evaluation is performed; that is, they can defer evaluation from occurring in the client code, or they can defer evaluation completely at the client side and let it take place in the server instead. Programmers who desire to defer evaluation to the client stub can write an evaluation routine that sets the client's and server's supported code sets into the binding handle returned by the client, then write the evaluation logic into the stub support routine for tag-setting so that it performs evaluation within the client stub.

Applications that do evaluation in the client stub take the chance that the binding handle that is evaluated is the only binding handle available. For example, suppose there are three binding handles. Two are character and code set compatible, and one is incompatible. The incompatible binding is selected for RPC. If you evaluate in the tag-setting routine, you cannot reselect to get the other compatible bindings.

In general, it is recommended that character and code sets evaluation take place in the client, rather than the server, for performance reasons. Also, once the server is selected and a connection is established between it and the client, the client cannot typically reselect the server because the code sets are incompatible.

Within the client, it is recommended that evaluation be performed in the client code rather than in client stub, because deferring evaluation to occur in the client stub removes any way for the client to gain access to other potential binding handles.

**Notes about Tag-Setting:** The DCE RPC character and code sets compatibility evaluation routines set the method and the code set tag values into a data structure in the binding handle returned to the client. These routines always set the sending tag and desired receiving tag to the same code set value.

In addition, if the application uses the DCE RPC routine **rpc_cs_get_tags()** to set the code set tags for the stubs, the value of the server's receiving tag will always be the value of what the client sent to it in the desired receiving tag. If RMIR is used, the desired receiving tag is the server's current code set.

RPC application programmers that do not want to use the DCE RPC-provided evaluation routines can use the **rpc_cs_binding_set_tags()** routine to set the code set tag values into a binding handle.

**Example Character and Code Sets Evaluation Logic:** Here is an example client program of the “cs_test” interface that provides its own character and code sets evaluation logic. This example client uses the **rpc_cs_binding_set_tags()** routine to set the code set tags within the client code rather than using a tag-setting routine to set them within the stub code.

```
#include <stdio.h>
#include <locale.h>
#include <dce/rpc.h>
#include <dce/rpcsts.h>
#include <dce/dce_error.h>

#include "cs_test.h"         /* IDL generated include file */

/*
 * Result check MACRO
 */
#define CHECK_STATUS(t, func, returned_st, expected_st) \
{ \
        if (returned_st == expected_st) { \
        ;  /* No operation */

        } else { \
                dce_error_inq_text(returned_st,\
                 (unsigned char *)unexpected, &dce_status); \
                dce_error_inq_text(expected_st,\
                 (unsigned char *)expected, &dce_status); \
                printf("FAILED %s()\nresult:   %s\nexpected:  %s\n\n", \
                                func, unexpected, expected); \
                if (t) \
                        return; \
                } \
        } \
} \

static unsigned char    unexpected[dce_c_error_string_len];
static unsigned char    expected[dce_c_error_string_len];
static int              dce_status;
```

```
void
main(void)
{
        rpc_binding_handle_t      bind_handle;
        rpc_ns_handle_t           lookup_context;
        rpc_binding_vector_p_t    bind_vec_p;
        unsigned_char_t           *entry_name;
        unsigned32                binding_count;
        cs_byte                   net_string[SIZE];
        cs_byte                   loc_string[SIZE];
        int                       i, k, rpc_num;
        int                       model_found, smir_true, cmir_true;
        rpc_codeset_mgmt_p_t      client, server;
        unsigned32                stag;
        unsigned32                drtag;
        unsigned16                stag_max_bytes;
        error_status_t            status;
        error_status_t            temp_status;
        unsigned char             err_buf[256];
        char                      nsi_entry_name[256];
        char                      *get_nsi_entry_name;
        char                      *client_locale_name;
        FILE                      *fp_in, *fp_out;

        get_nsi_entry_name = getenv("I18N_SERVER_ENTRY");
        strcpy(nsi_entry_name, get_nsi_entry_name);

        setlocale(LC_ALL, "");

        rpc_ns_binding_lookup_begin (
                rpc_c_ns_syntax_default,
                (unsigned_char_p_t)nsi_entry_name,
                cs_test_v1_0_c_ifspec,
                NULL,
                rpc_c_binding_max_count_default,
                &lookup_context,
                &status );

        CHECK_STATUS(TRUE, "rpc_ns_binding_lookup_begin", status, rpc_s_ok);

        rpc_ns_binding_lookup_next (
                lookup_context,
                &bind_vec_p,
                &status );

        CHECK_STATUS(TRUE, "rpc_ns_binding_lookup_next", status, rpc_s_ok);

        rpc_ns_binding_lookup_done (
                &lookup_context,
                &status );

        CHECK_STATUS(TRUE, "rpc_ns_binding_lookup_done", status, rpc_s_ok);

        /*
         *  Get the client's supported code sets
         */
        rpc_rgy_get_codesets (
                &client,
                &status );

        CHECK_STATUS(TRUE, "rpc_rgy_get_codesets", status, rpc_s_ok);

        binding_count = (bind_vec_p)->count;
        for (i=0; i < binding_count; i++)
```

```
{
        if ((bind_vec_p)->binding_h[i] == NULL)
                continue;

        rpc_ns_binding_select (
                bind_vec_p,
                &bind_handle,
                &status );

        CHECK_STATUS(FALSE, "rpc_ns_binding_select", status, rpc_s_ok);

        if (status != rpc_s_ok)
        {
                rpc_ns_mgmt_free_codesets(&client, &temp_status);
                return;
        }

        rpc_ns_binding_inq_entry_name (
                bind_handle,
                rpc_c_ns_syntax_default,
                &entry_name,
                &status );

        CHECK_STATUS(FALSE, "rpc_ns_binding_inq_entry_name", status, rpc_s_ok);
        if (status != rpc_s_ok)
        {
                rpc_ns_mgmt_free_codesets(&client, &temp_status);
                return;
        }

        /*
         *  Get the server's supported code sets from NSI
         */
        rpc_ns_mgmt_read_codesets (
                rpc_c_ns_syntax_default,
                entry_name,
                &server,
                &status );

        CHECK_STATUS(FALSE, "rpc_ns_mgmt_read_codesets", status, rpc_s_ok);

        if (status != rpc_s_ok)
        {
                rpc_ns_mgmt_free_codesets(&client, &temp_status);
                return;
        }

        /*
         *  Start evaluation
         */
        if (client->codesets[0].c_set == server->codesets[0].c_set)
        {
                /*
                 *  client and server are using the same code set
                 */
                stag = client->codesets[0].c_set;
                drtag = server->codesets[0].c_set;
                break;
        }

        /*
         *  check character set compatibility first
         */
        rpc_cs_char_set_compat_check (
                client, server, &status );

        CHECK_STATUS(FALSE, "rpc_cs_char_set_compat_check",
                 status, rpc_s_ok);

        if (status != rpc_s_ok)
        {
                rpc_ns_mgmt_free_codesets(&server, &temp_status);
                rpc_ns_mgmt_free_codesets(&client, &temp_status);
                return;
        }

        smir_true = cmir_true = model_found = 0;

        for (k = 1; k < server->count; k++)
        {
                if (smir_true)
                        break;

                if (client->codesets[0].c_set
                        == server->codesets[k].c_set)
                {
                        smir_true = 1;
                        model_found = 1;
                }
        }

        for (k = 1; k < client->count; k++)
        {
                if (cmir_true)
                        break;

                if (server->codesets[0].c_set
                        == client->codesets[k].c_set)
                {
                        cmir_true = 1;
                        model_found = 1;
                }


        }

        if (model_found)
        {
                if (smir_true && cmir_true)
                {
                        /* RMIR model works */
                        stag = client->codesets[0].c_set;
                        drtag = server->codesets[0].c_set;
                        stag_max_bytes
                             = client->codesets[0].c_max_bytes;
                }
                else if (smir_true)
                {
                        /* SMIR model */
                        stag = client->codesets[0].c_set;
                        drtag = client->codesets[0].c_set;
                        stag_max_bytes
                             = client->codesets[0].c_max_bytes;
                }
                else
                {
                        /* CMIR model */
                        stag = server->codesets[0].c_set;
```

```
                        drtag = server->codesets[0].c_set;
                        stag_max_bytes
                             = server->codesets[0].c_max_bytes;
                }

                /*
                 *  set tags value to the binding
                 */
                rpc_cs_binding_set_tags (
                        &bind_handle,
                        stag,
                        drtag,
                        stag_max_bytes,
                        &status );

                CHECK_STATUS(FALSE, "rpc_cs_binding_set_tags",
                        status, rpc_s_ok);
                if (status != rpc_s_ok)
                {
                        rpc_ns_mgmt_free_codesets(&server, &temp_status);
                        rpc_ns_mgmt_free_codesets(&client, &temp_status);
                        return;
                }
        }
        else
        {
                /*
                 *  try another binding
                 */
                rpc_binding_free (
                        &bind_handle,
                        &status );

                CHECK_STATUS(FALSE, "rpc_binding_free", status, rpc_s_ok);
                if (status != rpc_s_ok)
                {
                        rpc_ns_mgmt_free_codesets(&server, &temp_status);
                        rpc_ns_mgmt_free_codesets(&client, &temp_status);
                        return;
                }
        }
}

rpc_ns_mgmt_free_codesets(&server, &status);
CHECK_STATUS(FALSE, "rpc_ns_mgmt_free_codesets", status, rpc_s_ok);

rpc_ns_mgmt_free_codesets(&client, &status);
CHECK_STATUS(TRUE, "rpc_ns_mgmt_free_codesets", status, rpc_s_ok);

if (!model_found)
{
        printf("FAILED No compatible server found\n");
        return;
}

rpc_ep_resolve_binding (bind_handle,
        cs_test_v1_0_c_ifspec,
        &temp_status);

CHECK_STATUS(TRUE, "rpc_ep_resolve_binding", temp_status, rpc_s_ok);

if(rpc_mgmt_is_server_listening(bind_handle, &status)
        && temp_status == rpc_s_ok)
{
```

```
                printf("PASSED rpc_mgmt_is_server_listening()\n");
        }
        else
        {
                dce_error_inq_text ((unsigned long)status, err_buf,
                 (int *)&temp_status);
                printf("is_server_listening error -> %s\n", err_buf);
        }

        fp_in = fopen("./i18n_input_data", "r");

        if (fp_in == NULL)
        {
                printf("i18n_input_data open failed\n");
                return;
        }

        fp_out = fopen("./i18n_tags_fixed_result_file", "w");

        if (fp_out == NULL)
        {
                printf("i18n_result_file open failed\n");
                return;
        }

        rpc_num = 1;
        (void)fgets((char *)net_string, SIZE, fp_in);

        while (!feof(fp_in))
        {
                temp_status = cs_fixed_trans(bind_handle, net_string, loc_string);

                if (temp_status != rpc_s_ok)
                {
                        dce_error_inq_text(temp_status, err_buf, (int *)&status);

                        printf("FAILED %ld  MSG: %s\n", (unsigned long)temp_status, err_buf);
                }
                else
                {
                        printf("PASSED rpc #%d\n", rpc_num++);
                        (void)fputs((char *)loc_string, fp_out);
                        (void)fputs("\n", fp_out);
                }
                (void)fgets((char *)net_string, SIZE, fp_in);
        }

        fclose(fp_in);
        fclose(fp_out);

        return;
}
```

# Chapter 10. Topics in RPC Application Development

This chapter describes special features of DCE RPC for application development. The topics include:

- Memory management
- Error handling
- Context handles
- Pipes
- Nested calls and callbacks
- Routing RPCs
- Portable data and the IDL encoding services

## Memory Management

When called to handle a remote operation, RPC client stubs allocate and free memory using whatever memory management scheme is currently in effect. Client code, that is, generic code that can be called from either RPC clients or RPC servers, can use DCE RPC stub support routines to control which memory management scheme the stubs will use.

If client code has not explicitly set the memory management routines, the RPC client stubs use the following defaults:

- When called from manager code, and the operation contains one or more parameters that are full or unique pointers, or the ACF attribute enable_allocate has been applied, the client stubs use the **rpc_ss_allocate()** and **rpc_ss_free()** routines.
- When called from any other context, the RPC client stubs use the operating system allocation and free routines, for example **malloc()** and **free()** on POSIX platforms.

Note that the memory management scheme established, whether explicitly or by default, is on a per-thread basis.

RPC server stubs do not allocate memory. Instead, they rely on the manager code, that is, the code that the server stubs call, to allocate it for them.

The following sections give guidelines for how client code and manager code should use the various allocation and free routines provided with DCE.

**Note:** DCE provides two versions of DCE RPC stub support routines. The **rpc_ss_** routines raise an exception, while the **rpc_sm** routines return an error status value. In all other ways, the routines are identical. It is generally recommended that you use the **rpc_sm** routines instead of the **rpc_ss** routines for compliance with Application Environment Specification for DCE RPC.

## Using the Memory Management Defaults

If it does not matter to the client code which memory allocation routine the RPC client stubs use, the client code should call the **rpc_ss_client_free()** routine to free any memory that the client stub allocates and returns. The **rpc_ss_client_free()** routine uses the current free routine that is in effect. Client code that uses **rpc_ss_client_free()** must use caution if it calls other routines before it frees all of the pieces of allocated storage with **rpc_ss_client_free()**, because it is possible that the called code has been written so that it swaps in a different allocation/free pair without re-establishing the previous allocation/free pair on exit.

## Using rpc_ss_allocate and rpc_ss_free

Both client code and manager code can use **rpc_ss_allocate()** and **rpc_ss_free()**. The following sections describe how.

**Using rpc_ss_allocate and rpc_ss_free in Manager Code:** Manager code uses either the **rpc_ss_allocate()** and **rpc_ss_free()** routines or the operating system allocation and free routines to allocate and free memory.

Manager code uses **rpc_ss_allocate()** to allocate storage for data that the server stub is to send back to the client. Manager code can either use **rpc_ss_free()** to free the storage explicitly, or it can rely on the server stub to free it. After the server stub marshalls the output parameters, it releases any storage that the manager code has allocated with **rpc_ss_allocate()**.

Manager code can also use the **rpc_ss_free()** routine to release storage pointed to by a full pointer in an input parameter and have the freeing of the memory reflected on return to the calling application if the **reflect_deletions** attribute has been specified as an operation attribute in the interface definition. See Chapter 11, “Interface Definition Language” on page 221 for instructions on how to declare the **reflect_deletions** operation attribute.

Manager code uses the operating system allocation routine to create storage for its internal data. The server stub does not automatically free memory that operating system allocation routines have allocated. Instead, manager code must use the operating system free routine to deallocate the memory explicitly before it exits.

When manager code makes a remote call, the default memory management routines are **rpc_ss_allocate()** and **rpc_ss_free()**.

**Using rpc_ss_allocate and rpc_ss_free in Client Code:** Client code may also want to use the **rpc_ss_allocate()** and **rpc_ss_free()** routines as the stub memory management scheme. However, before client code can use **rpc_ss_allocate()** and **rpc_ss_free()**, it must first call the **rpc_ss_enable_allocate()** routine, which enables the use of **rpc_ss_allocate()**. If client code calls **rpc_ss_enable_allocate()**, it must also call the **rpc_ss_disable_allocate()** routine before it exits its thread to disable use of **rpc_ss_allocate()**. This routine releases all of the memory allocated by calls to **rpc_ss_allocate()** in that thread since the call to **rpc_ss_enable_allocate()** was made. As a result, client code can either free each piece of allocated storage with **rpc_ss_free()**, or it can have **rpc_ss_disable_allocate()** free it all at once when it disables the **rpc_ss_allocate/free** memory management scheme.

Before calling **rpc_ss_enable_allocate()**, client code must ensure that it has not been called by code that has already set up the **rpc_ss_allocate/free** memory management scheme. As a result, if the client code can ensure that it has not been called from a manager routine, *and* it can ensure that any previous calls to

**rpc_ss_enable_allocate()** have been paired with calls to **rpc_ss_disable_allocate()**, it can safely call **rpc_ss_enable_allocate()**.

If client code cannot ensure that these conditions are true, it should check to make sure the **rpc_ss_allocate/free scheme** has not already been set up. For example:

```
/* Get RPC memory allocation thread handle */

    rpc_ss_thread_handle_t thread_handle;
    idl_void_p_t (*p_saved_alloc)(unsigned long);
    void (*p_saved_free)(idl_void_p_t);

    TRY
        thread_handle = rpc_ss_get_thread_handle();
    CATCH(pthread_badparam_e)
        thread_handle = NULL;
    ENDTRY

    if (thread_handle == NULL)    {

     /* Set up rpc_ss_allocate environment */

        rpc_ss_enable_allocate();
    }

    rpc_ss_swap_client_alloc_free(
        appl_client_alloc,appl_client_free,
        &p_saved_alloc,&p_saved_free);
```

After control returns from the client stub, the client code should again check to see whether **rpc_ss_allocate/free** has already been enabled before it calls **rpc_ss_disable_allocate()**:

```
    rpc_ss_set_client_alloc_free(p_saved_alloc,p_saved_free);

/* If we set up rpc_ss_allocate environment, disable it now */

    if (thread_handle == NULL)
        rpc_ss_disable_allocate();
```

## Using Your Own Allocation and Free Routines

At times it might be necessary for client code to change the routines that the client stubs use to allocate and free memory. For example, client code that is making an RPC call might want to direct the RPC client stubs to use special debug versions of **malloc()** and **free()** that check for memory leaks. Another example might be an application that uses DCE RPC but needs to preserve its users' ability to free memory returned from the application using the platform's memory management scheme (rather than exposing the user to DCE).

Client code that wants to use its own memory allocation and free routines can use the **rpc_ss_swap_client_alloc_free()** routine to exchange the current client allocation and freeing mechanism for one supplied in the call. The routine returns pointers to the memory allocation and free routines formerly in use. Before calling **rpc_ss_swap_client_alloc_free()**, client code must ensure that it has not been called from a manager routine.

Deallocation of allocated storage returned from the client stubs is not automatic. Therefore, client code must ensure that it uses the free routine that it specified in the call to to **rpc_ss_swap_client_alloc_free()** to deallocate each piece of allocated storage.

Client code that swaps in memory management routines with **rpc_ss_swap_client_alloc_free()** should use the **rpc_ss_set_client_alloc_free()** routine before it exits to restore the old allocation and free routines.

## Using Thread Handles in Memory Management

There are two situations where control of memory management requires the use of thread handles. The more common situation is when the manager thread spawns additional threads. The less common situation is when a program transitions from being a client to being a server, then reverts to being a client.

**Spawning Threads:** When a remote procedure call invokes the manager code, the manager code may wish to spawn additional threads to complete the task for which it was called. To spawn additional threads that are able to perform memory management, the manager code must first call the **rpc_ss_get_thread_handle()** routine to get its thread handle and then pass that thread handle to each spawned thread. Each spawned thread must call the **rpc_ss_set_thread_handle()** routine with the handle received from the manager code.

These routine calls allow the manager and its spawned threads to share a common memory management environment. This common environment enables memory allocated by the spawned threads to be used in returned parameters and causes all allocations in the common memory management environment to be released when the manager thread returns to the server stub.

The main manager thread must not return control to the server stub before all the threads it spawned complete execution; otherwise, unpredictable results may occur.

The listener thread can cancel the main manager thread if the remote procedure call is orphaned or if a cancelation occurs on the client side of the application. You should code the main manager thread to terminate any spawned threads before it exits. The code should anticipate exits caused by an unexpected exception or by being canceled.

Your code can handle all of these cases by including a TRY/FINALLY block to clean up any spawned threads if a cancelation or other exception occurs. If unexpected exceptions do not concern you, then your code can perform two steps. They are disabling cancelability before threads are spawned followed by enabling cancelability after the join operation finishes and after testing for any pending cancel operations. Following this disable/enable sequence prevents routine **pthread_join()** from producing a cancel point in a manager thread that has spawned threads which, in turn, share thread handles with the manager thread.

**Transitioning from Client to Server to Client:** Immediately before the program changes from a client to a server, it must obtain a handle on its environment as a client by calling **rpc_ss_get_thread_handle()**. When it reverts from a server to a client, it must reestablish the client environment by calling the **rpc_ss_set_thread_handle()** routine, supplying the previously obtained handle as a parameter.

## Guidelines for Error Handling

During a remote procedure call, server and communications errors may occur. You can handle them using any or all of the following methods:

- Writing exception handler code to recover from the error or to exit the application
- Using the **fault_status** attribute in the ACF to report an RPC server failure
- Using the **comm_status** attribute in the ACF to report a communications failure.

Use of exceptions, where the procedure exits the program because of an error, tends to improve code quality. Exceptions make errors more visible because of the point where the program exits, and reduces the amount of code needed to detect error conditions and handle them. When you use the **fault_status** or **comm_status** IDL attribute, an exception that occurs on the server is not reported to the client as an exception. The variable to which the **fault_status** or **comm_status** attribute is attached, contains error codes. These codes report errors that would not have occurred if the application were not distributed over a communications network. The **fault_status** and **comm_status** attributes provide a method of handling RPC errors without using an exception handler.

## Exceptions

Exceptions report errors, either RPC errors or errors in application code, when **comm_status** or **fault_status** or both are not present in the ACF. If you use exceptions, you:

- Do not have to adjust procedure declarations between local and distributed code.
- Can distribute existing interfaces without changing code.
- Do not have to check for failures. This results in more versatile code because errors are reported even if they are not checked.
- Write more efficient code than when there is no recovery coded for failures.
- Can use a simpler coding style.

Exceptions work well for coarse-grained exception handling. If your application does not contain any exception handlers and the application thread gets an error, the application thread is ended and a system-dependent error message from the threads package is printed.

**Note:** RPC exceptions are equivalent to RPC status codes. To identify the status code that corresponds to a given exception, replace the **_x_** string of the exception with the string **_s_**. For example, the exception **rpc_x_comm_failure** is equivalent to the status code **rpc_s_comm_failure**. The RPC exceptions are defined in the **<dce/rpcexc.h>** header file.

The RPC status codes are documented in *z/OS DCE Messages and Codes*. The documentation for each status code includes the message text, the explanation, and the suggested user action.

The set of exceptions that can always be returned from the server to the client (such as the **rpc_x_invalid_tag** exception) are referred to as *system exceptions*. These exceptions are defined in **dce/rpcexc.h** and **dce/exc_handling.h**.

An interface definition can also specify a set of user-defined exceptions that the interface's operations can return to the client. You can declare user-defined exceptions in an interface definition by using the **exceptions** interface attribute, which is described in Chapter 11, “Interface Definition Language” on page 221.

If a user-defined exception in the implementation of a server operation occurs during server execution, the server terminates the operation and propagates the exception to the client in a manner similar to the way system exceptions are propagated. If a server implementation of an operation raises an exception that is neither a system exception nor a user-defined exception, the exception returned to the client is **rpc_x_unknown_remote_fault**.

By default, the IDL compiler defines and initializes all exceptions under a "once block" in the generated stubs. If you want to share exception names in multiple interfaces or you desire greater control over how these exceptions are defined and initialized, you can use the ACF **extern_exceptions** attribute to disable the automated mechanism that the IDL compiler uses to define and initialize exceptions. See Chapter 12, “Attribute Configuration Language” on page 285 for more information on the **extern_exceptions** attribute.

Because exceptions are associated with operation implementation, they are not imported into other interfaces by way of the **import** declaration. For more information about using exceptions to handle errors, see Part 3, “Using the DCE Threads APIs” on page 313.

## The fault_status Attribute

The **fault_status** attribute requests that errors occurring on the server due to incorrectly specified parameter values, resource constraints, or coding errors be reported by an additional status parameter instead of by an exception. If a user-defined exception is returned from a server to a client that has specified **fault_status** on the operation in which the exception occurred, the value given to the **fault_status** parameter is **rpc_s_fault_user_defined**.

The **fault_status** attribute has the following characteristics:

- Occurs where you do not want transparent local/remote behavior
- Occurs where you expect that you may be passing incorrect data to the server or the server is not coded robustly, or both
- Works well for fine-grained error handling
- Requires that you adjust procedure declarations between local and distributed code
- Controls the reporting only of errors that come from the server and that are reported via a fault packet

For more information on the **fault_status** attribute, see Chapter 12, “Attribute Configuration Language” on page 285.

## The comm_status Attribute

The **comm_status** attribute requests that RPC communications failures be reported through an additional status parameter instead of by an exception. The **comm_status** attribute has the following characteristics:

- Occurs where you expect communications to fail routinely; for instance, no server is available, the server has no resources, and so on
- Works well for fine-grained error handling; for example, trying a procedure many times until it succeeds
- Requires that you adjust procedure declarations between local and distributed code to add the new status parameter
- Controls the reporting of errors only from RPC runtime error status codes

For more information on the **comm_status** attribute, see Chapter 12, “Attribute Configuration Language” on page 285.

## Determining Which Method to Use for Handling Exceptions

Some conditions are better for using the **comm_status** or **fault_status** attribute on an operation, rather than the default approach of handling exceptions.

The **comm_status** attribute is useful only if the call to the operation has a specific recovery action to perform for one or more communications failures; for example, **rpc_s_comm_failure** or **rpc_s_no_more_bindings**. The **comm_status** attribute is recommended only when the application knows that it is calling a remote operation. If you expect communications to fail often because the server does not have enough resources to execute the call, you can use this attribute to allow the call to be retried several times. If you are using an implicit or explicit binding, you can use the **comm_status** attribute if you want to try another server because the operation cannot be performed on the one you are currently using. You can also use an exception handler for each of the two previous instances. In general, the advantage of using **comm_status** is that the recovery is local to the routine and so the overhead is less. The disadvantage of using **comm_status** is that it results in two different operation signatures. Distributed calls contain the **comm_status** attribute, however; local calls do not. Also, if all of the recovery cannot be done locally (where the call is made), there must be a way to pass the status to outer layers of code to process it.

The **fault_status** attribute is useful only if the call to the operation has a specific recovery action to perform for one or more server faults; for example, **rpc_s_invalid_tag**, **rpc_s_fault_pipe_comm_error**, **rpc_s_fault_int_overflow**, or **rpc_s_fault_remote_no_memory**. Use **fault_status** only when the application calls a remote operation and wants different behavior than if it calls the same operation locally. If you are requesting an operation on a large data set you can use this attribute to trap **rpc_s_fault_remote_no_memory** and retry the operation to a different server, or you may break your data set into two smaller sections. You can also handle the previous case with exception handlers. The advantage of using **fault_status** if the recovery is local is that the overhead is less. The disadvantage of **fault_status** is that the operation is different between the local and distributed case. Also, if all of the recovery cannot be done locally, there must be a way to pass the status to outer layers of code to process it.

## Examples of Error Handling

The following sections present two examples of error handling. The first example assumes that the **comm_status** attribute is in use in the ACF. The second example assumes that the **comm_status** attribute is not in use.

**The Matrix Math Server Example:** Assume that you have an existing local interface that provides matrix math operations. Since it is local, errors such as floating-point overflow or divide by zero are returned to the caller of a matrix operation as exceptions. It is likely that these exceptions are caused by providing data to the operation in an improper form.

In this case, the exceptions are part of the interface, so **fault_status** changes the way the application calls the matrix interface and probably is undesirable. Depending on the environment, finding a server may not be difficult (if the network is relatively stable and has enough resources), and adding **comm_status** serves only to introduce differences between the local and distributed applications.

If a decision as to what action to take is based upon a communications failure, then you may try to add the conditional code **comm_status** requires. Otherwise, using **auto_handle** allows an attempt on each available server. If no server is available, the application terminates because it cannot proceed. You can add an exception handler to the main program to report the error in a user-friendly manner.

**The Stock Quote Application Example:** Assume that you have a Windows application that reads from stock quote servers and displays graphs of the data. Since you do not expect to get server failures because it is a commercial-quality server, you are not interested in writing code to handle values returned from **fault_status**. If high availability and robustness is important, you may have a list of recovery plans to make sure a stock analyst can get the necessary information as quickly as possible. For example:

```
retry_count = 10;
do
    query_stock_quote(h, ...,&st);
    switch (st)        /* st parameter can be used because */
    {                   /* [comm_status] is in the ACF */
        case rpc_s_ok:
            break;
        case rpc_s_comm_failure:
            retry_count -= 1;
            break;
        case rpc_s_network_unreachable:
            h = some_other_handle;
            break;
        case
            .
            .
            .
        default:
            retry_count -= 1;
    }
while ((st == rpc_s_ok) || (retry_count <= 0))
```

If this is not a critical application, you may only report that the server is currently unavailable. Depending upon the design of the application, there may be several places to put the exception handler to report the failure but continue processing. For example:

```
TRY
    update_a_quote(...);
CATCH_ALL
    display_message("Stock quote not currently available");
ENDTRY
```

This example assumes that **update_a_quote()** eventually calls the remote operation **query_stock_quote()** and that this call may raise an exception that is detected and reported here.

The advantage of using exceptions in this case is that all of the work done in **update_a_quote()** has the same error recovery and it does not need to be repeated at every call to a remote operation. Another advantage is that if one of the remote operations does have a recovery for one exception, it can handle that one exception and allow the rest to propagate to the more general handler in an outer layer of the code.

## Context Handles

During a series of remote procedure calls, the client may need to refer to a context maintained by a specific server instance. Server application code can maintain information it needs for a particular client (such as the state of RPC the client is using) as a context. To provide a client with a means of referring to its context, the client and server pass back and forth an RPC-specific parameter called a *context handle*. A context handle is a reference (a pointer) to the server instance and the context of a particular client. A context handle ensures that subsequent remote procedure calls from the client can reach the server instance that is maintaining context for the client.

On completing the first procedure in a series, the server passes a context handle to the client. The context handle identifies the context that the server uses for subsequent operations. The client is not supposed to do anything with the context handle; it merely passes it to subsequent calls as needed, and it is used internally by the remote calls. This allows applications to have such things as remote calls that handle file operations much as local calls would; that is, a client application can remotely open a file, get back a handle to it, and then perform various other remote operations on it, passing the context handle as an argument to the calls. A context handle can be used across interfaces (where a single server offers the multiple interfaces), but a context handle belongs only to the client who caused it to be activated.

The server maintains the context for a client until the client calls a remote procedure that terminates use of the context or communications are lost. In the latter case, the server's runtime can invoke a context rundown procedure. This application-specific routine is called by the server stub automatically to reclaim (rundown) the pointed-to resource in the event of a communications break between the server and client. For example, in the case of the remote file pointer just mentioned, the context rundown routine would simply close the file.

As usual with RPC, you need to apply indirection operators in a variety of ways to maintain the correct **[in]** and **[out]** semantics. Typical declarations for a context handle follow.

In the **.idl** file, declare a named type such as:

```
typedef [context_handle] void* my_handle_t;
```

A manager routine that returns a context handle as an **out** parameter declares it as:

```
my_handle_t *h;
```

The routine then sets the value of the handle as follows:

```
*h = &context_data;
```

A routine that refers to a context handle as an **in** parameter declares it as:

```
my_handle_t h;
```

and dereferences the handle as follows:

```
context_data = (my_handle_t*)h;
```

For the **in,out** case, the routine uses the same declaration as in the **out** case, and dereferences the handle as follows:

```
context_data = (my_handle_t*)*h;
```

The following extensive example shows a simple use of context handles. In the sample code, the client requests a unit of storage from the server, using the **store_open()** call, and receives a handle to the allocated storage. The **store_read()**, **store_write()**, and **store_set_ptr()** routines allow the client to read from and write to specific locations in the allocated storage. The **store_close()** routine releases the server resources.

## Context Handles in the Interface

The **.idl** file declarations for the **store** interface are as follows:

```
/*
 * store.idl
 * A sample interface that demonstrates server maintained context.
 * The client requests temporary storage of a specified size,
 * and the server returns a handle that can be used to read and
 * write to storage. The interface doesn't care how the
 * server implements the storage.
 */
[
uuid(0019b8c5-e8b5-1c84-9a41-0000c0d4de56),
pointer_default(ref),
version(1.0)
]
interface store
{
        /* A context handle used to access remote storage:                 */
        typedef [context_handle] void* store_handle_t;
        /* A storage object name string:                                */
        /*  typedef [string] char* store_name_t; */

        /* A buffer type for data:                                      */
        typedef byte store_buf_t[*];

        /* Note that the context handle is an [out] parameter of the open   */
        /*  routine, an [in, out] parameter of the close routine, and an    */
        /*  [in] parameter of the other routines. If the context handle     */
        /*  were treated as an [in] parameter of the close routine, the     */
        /*  stubs would never learn that the context had been set to NULL,  */
        /*  and would consider the context to still be live. This would     */
        /*  result in the rundown routine's being called when the client    */
        /*  terminated, even though there would be no context to run down.  */

        void store_open(
           [in] handle_t binding,
           [in] unsigned32 store_size,
           [out] store_handle_t *store_h,
           [out] error_status_t *status
        );

        void store_close(
           [in,out] store_handle_t *store_h,
           [out] error_status_t *status
        );

        void store_set_ptr(
           [in] store_handle_t store_h,
           [in] unsigned32 offset,
           [out] error_status_t *status
        );

        void store_read(
           [in] store_handle_t store_h,
           [in] unsigned32 buf_size,
           [out, size_is(buf_size), length_is(*data_size)] store_buf_t buffer,
           [out] unsigned32 *data_size,
           [out] error_status_t *status
        );

        void store_write(
           [in] store_handle_t store_h,
           [in] unsigned32 buf_size,
           [in, size_is(buf_size)] store_buf_t buffer,
           [out] unsigned32 *data_size,
           [out] error_status_t *status
        );
}
```

## Context Handles in a Server Manager

Server manager code to provide a rudimentary implementation of the **store** interface is as follows:

```
/* context_manager.c -- implementation of "store" interface.             */
/*                                                                        */
/*                                                                        */
/*   The server maintains a certain number of storage areas, only one of  */
/*    which can be (or should be) opened by a single client at a time.    */
/*    More than one client can, however, apparently be invoked (up to the */
/*    number of separate storelets == store handles available, defined by */
/*    the value of NUM_STORELETS). Each client keeps track of its store   */
/*    (and likewise enables the server to do the same) by means of the con- */
/*    text handle it receives when it opens its store.                     */
/*                                                                          */
/*                                                                          */
/*                                                                          */
/*                           -77 cols-                                      */
/****************************************************************************/

#include <ibm/dce>
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <pthread.h>
#include <dce/dce_error.h>
#include <dce/daclif.h>

#include "context.h"

#define NUM_STORELETS 10

/****************************************************************************/
/* The actual "storelet" structure...                                       */

typedef struct store_hdr{
        pthread_mutex_t ref_lock;
        unsigned32 size;
        unsigned32 refcount;
        idl_byte *storage;
} store_hdr_t;

store_hdr_t headers[NUM_STORELETS]; /* There's an array of these.          */

/****************************************************************************/
/* The store specification structure; note that it is equivalent to the     */
/*  handle; the pointer to it is returned as the handle by the store_open() */
/*  routine below...                                                        */
/*  The assumption is that all access to a given handle is serialized in a  */
/*  single thread, so no locking is needed for these.                       */

typedef struct store_spec{
        unsigned32 number;     /* The storelet number we've opened.         */
        unsigned32 offset;     /* The current read/write position.          */
} store_spec_t;  /* There is only one of these; it's the handle that gives  */
                 /*  access to one of the NUM_STORELETS set of "storelets". */


/* The server entry name:                                                   */
extern unsigned_char_p_t entry;


/* Initialization control block:                                            */
pthread_once_t init_once_blk = pthread_once_init;




/******
*
*
* store_mgmt_init -- Zeroes out all the storelet structures; executed only
*                    once per server instance, as soon as a client has
*                    called the store_open() routine.
*
*
******/
```

```
/***************************************************************************/
void
store_mgmt_init(
)
{
        int i;
        store_hdr_t *hdr;

        fprintf(stdout, "Store Manager: Initializing Store\n");
        memset(headers, 0, sizeof(store_hdr_t) * NUM_STORELETS);
        for (i = 0; i < NUM_STORELETS; i++)
        {
                hdr = headers + i;
                pthread_mutex_init(
                        (pthread_mutex_t *)hdr,
                        pthread_mutexattr_default);
        }

}





/******
*
*
* store_open -- Opens a store and returns a handle to it. The store consists
*               of one "storelet" selected from the array of NUM_STORELETS.
*
*
******/
/***************************************************************************/
void
store_open(
        handle_t binding,
        unsigned32 store_size,    /* Size specified for actual storage.     */
        store_handle_t *store_h,  /* To return the store handle in.         */
        error_status_t *status
)
{
        int i;                 /* Index variable.                           */
        store_spec_t *spec;    /* Store specification == handle.            */
        store_hdr_t *hdr;      /* Storelet structure.                       */

        /* Do the store initialization if this is the first open call...    */
        /* Zero out the store headers...                                    */
        pthread_once(&init_once_blk, store_mgmt_init);

        /* The following loop goes through all the storelets, looking for   */
        /*  one whose reference count is zero. As soon as one such is       */
        /*  found, a handle is allocated for it, storage is allocated for   */
        /*  its store structure, and the loop (and the call) terminates. If */
        /*  no unreferenced storelet is found, a status of -1 is returned   */
        /*  and no handle is allocated...                                   */
        for(i = 0; i < NUM_STORELETS; i++)
        {
                /* Go to the next storelet...                               */
                hdr = headers + i;

                /* Is it unreferenced?...                                   */
                if (hdr->refcount == 0)
                {
                        /* If so, lock the header...                        */
                        *status = pthread_mutex_lock((pthread_mutex_t *)hdr);
                        if (*status != 0)
                        {
                                return;
                        }

                        /* ... and check the reference count again...        */
                        if (hdr->refcount == 0)
                        {
                                /* Now we know we "really" have this one.   */
                                /* Only one open is allowed, so lock only   */
                                /*  the reference count...                  */
                                hdr->refcount++;

                                /* Now unlock the header so other threads   */
                                /*  can continue to check it...             */
                                *status = pthread_mutex_unlock((pthread_mutex_t *)hdr);
                                if (*status != 0)
                                        return;

                                /* Now allocate space for the specifica-    */
                                /*  tion structure...                       */
                                spec = (store_spec_t *)malloc(sizeof(store_spec_t));
                                spec->number = i;
                                spec->offset = 0;
                                *store_h = spec;

                                /* Allocate space for the storage part of   */
                                /*  the header...                           */
                                hdr->storage = (idl_byte *)malloc(store_size);
                                hdr->size = store_size;

                                /* Finally, set the return status to OK,    */
                                /*  and return...                           */
                                *status = error_status_ok;
                                return;
                        }

                        /* If the reference count turned out to have        */
                        /*  been accessed between our first check and our   */
                        /*  locking the mutex, we must now unlock the mutex */
                        /*  preparatory to looping around to check the next */
                        /*  storelet...                                     */
                        *status = pthread_mutex_unlock((pthread_mutex_t *)hdr);
                        if (*status != 0)
                        {
                                return;
                        }
                }
        }

        /* The following is reached only if we never found a free           */
        /*  storelet...                                                     */
        *store_h = NULL;
        *status = -1;

}



/******
*
*
```

```
* store_set_ptr -- Insert a new value into the store buffer pointer.
*
*
*******/
/****************************************************************************/
void store_set_ptr(
        store_handle_t store_h,     /* The store handle.                    */
        unsigned32 offset,    /* Value to insert into store buffer pointer. */
        error_status_t *status
)
{
        store_spec_t *spec;              /* Our pointer to store handle.    */

        spec = (store_spec_t *)store_h;  /* Get the store spec.             */
        spec->offset = offset;     /* Copy in the new buffer pointer value. */
        *status = error_status_ok;
}



/******
*
*
* store_close -- Close the opened storelet.
*
*
******/
/****************************************************************************/
void
store_close(
        store_handle_t *store_h,                /* Store handle.            */
        error_status_t *status
)
{
        store_spec_t *spec;             /* Our pointer to store handle.     */
        store_hdr_t *hdr;               /* Pointer to a storelet.           */

        printf("Store Manager: Closing Store\n");

        spec = (store_spec_t *)*store_h; /* Get the store spec.             */
        hdr = headers + spec->number;    /* Point to the correct storelet.  */

        /* If the thing is actually opened, close it...                     */
        if (hdr->refcount > 0)
        {
                /* Lock the header first...                                 */
                *status = pthread_mutex_lock((pthread_mutex_t *)hdr);
                if (*status != 0)
                {
                        printf("Close: lock failed\n");
                        return;
                }

                /* Check the reference count to make sure no one slipped in */
                /*  before we could lock the header, and already closed the */
                /*  storage...                                              */
                if (hdr->refcount > 0)
                {
                        /* The store is open, and it's locked by us, so we  */
                        /*  can safely close it. So do it. First, decrement */
                        /*  the reference count...                          */
                        hdr->refcount--;

                        /* Is it completely closed now?                     */
                        if (hdr->refcount == 0)
                        {
                                /* If so, get rid of its storage space...   */
                                hdr->size = 0;
                                free(hdr->storage);
                        }
                }

                /* If the store turned out to be closed before we could     */
                /*  close it, we have nothing to do but release the lock... */
                *status = pthread_mutex_unlock((pthread_mutex_t *)hdr);
                if (*status != 0)
                {
                        printf("Close: unlock failed\n");
                        return;
                }
        }

        /* And free our handle space...                                     */
        free(spec);

        /* Be sure to NULL the context handle.  Otherwise, the context      */
        /*  will be considered to be live as long as the client is run-     */
        /*  ning...                                                         */
        *store_h = NULL;
        *status = error_status_ok;
}



/******
*
*
* store_read -- Read a certain number of bytes from the opened store.
*
*
******/
/*****************************************************************************/
void
store_read(
        store_handle_t store_h,   /* Store handle.                          */
        unsigned32 buf_size,      /* Number of bytes to read.               */
        store_buf_t buffer,       /* Space to return data read in.          */
        unsigned32 *data_size,    /* To return number of bytes read in.     */
        error_status_t *status
)
{
        store_spec_t *spec;       /* Our handle pointer.                    */
        store_hdr_t *hdr;         /* Pointer to a storelet.                 */

        spec = (store_spec_t *)store_h;  /* Get the storelet spec.          */
        hdr = headers + spec->number;    /* Point to the correct storelet.  */

        /* If the amount we're to read is less than the amount left to be   */
        /*  read, then read it...                                           */
        if (buf_size <= hdr->size)
        {

                /* Copy bytes from the storelet storage, beginning at off-  */
                /*  set, into the return buffer, up to the size of the      */
                /*  buffer...                                               */
                memcpy(buffer, hdr->storage + spec->offset, buf_size);

                /* Update the storelet buffer pointer past what we've just  */
                /*  read...                                              */
                spec->offset += buf_size;

                /* Show return size of data read...                      */
                *data_size = buf_size;
                *status = error_status_ok;
                return;
        }

        /* If there's less data left than has been specified to read, don't */
        /*  read it...                                                   */
        *data_size = 0;
        *status = -1;
}


/******
*
*
* store_write -- Write some data into the opened store.
*
*
******/

void
store_write(
        /* handle_t IDL_handle,*/  /* If the server ACF declares            */
                                        /*  [explicit_handle]               */
        store_handle_t store_h,    /* Store handle.                         */
        unsigned32 buf_size,       /* Number of bytes to write.             */
        store_buf_t buffer,        /* Data to be written.                   */
        unsigned32 *data_size,     /* To return number of bytes written.    */
        error_status_t *status
)
{
        store_spec_t *spec;            /* Our pointer to store handle.      */
        store_hdr_t *hdr;              /* Pointer to a storelet.            */

        /* If the server ACF declares IDL handle as an explicit_handle,     */
        /* an access check on IDL handle could be done here.  See           */
        /* the example under "Binding and Security Information".            */

        spec = (store_spec_t *)store_h;  /* Get the storelet spec.          */
        hdr = headers + spec->number;    /* Point to the correct storelet.  */

        /* If the amount of unused room left in the storelet is greater     */
        /*  than what we're supposed to write in it, write it...            */
        if ((hdr->size - spec->offset) > buf_size)
        {

                /* Copy bytes from the buffer into the storelet storage,    */
                /*  beginning at the current read/write position...         */
                memcpy(hdr->storage + spec->offset, buffer, buf_size);

                /* Update the storelet buffer pointer to point past what    */
                /*  we've just written...                                   */
                spec->offset += buf_size;

                /* Add a null in case we want to read the store as a        */
                /*  string...                                               */
                *(hdr->storage + spec->offset) = 0;

                /* Show return size of data written...                      */
                *data_size = buf_size;
                *status = error_status_ok;
                return;
        }

        /* If we don't have room to write the whole buffer, don't write     */
        /*  anything...                                                     */
        *data_size = 0;
        *status = error_status_ok;
}


/******
 *
 * print_manager_error-- Manager version. Prints text associated with bad status code.
 *
 *
 *
 ******/
void
print_manager_error(
char *caller,  /* String identifying the routine that received the error.   */
error_status_t status) /* the status we want to print the message for.      */
{
        dce_error_string_t error_string;
        int print_status;

        dce_error_inq_text(status, error_string, &print_status);
        fprintf(stderr," Manager: %s: %s\n", caller, error_string);

}
```

The sample implementation of the store interface is obviously too limited for any practical use, but it does demonstrate the application of context handles in a straightforward way. A context handle returned by the **store_open()** routine is opaque to the client. To the server it is a pointer to the server's representation of a storage unit. In this case, it points to a structure that keeps track of the client's current location within a specific piece of server maintained storage.

Aside from deallocating the actual storage, the **store_close()** routine sets the context handle to **NULL**. The **NULL** value indicates to the server stub that the context is no longer active, and the stub, in turn, tells the RPC runtime not to maintain the context. For example, after the **store_close()** routine has been invoked, the rundown routine will not be invoked if communication ends between client and server. The context rundown routine takes care of closing the client's storage in case of a communication failure while the context is still active.

The global array of **store_hdr** structures that keeps track of allocated storage, obviously serves no practical purpose in the example. (Presumably the operating system is already doing this!) However, it does provide a demonstration of the fact that global server manager data is shared data in the implicitly multithreaded server environment. The routines that manipulate this shared data may be called simultaneously by multiple server threads (in response to multiple simultaneous client calls); therefore locking must be provided, in this case on the *refcount* field. The sample also demonstrates how the **pthread_once()** facility can be used to provide one-time initialization of the shared data on the first **store_open()** call.

As an exercise, the storage interface can easily be made more interesting by providing multiple clients simultaneous access to a given storage area. To implement this, the application could add a *store_name* parameter to the **store_open()** routine, and replace the *refcount* field with counts of readers and writers. The division of the storage management between the *store_hdr* and the *store_spec* data structures is intended to facilitate this; the *store_hdr* holds shared state relating to each store, while the *store_spec* holds each thread's private state.

## Context Rundown

Context handles typically point to some state maintained by a server instance for a client over a series of RPC operations. If the series of operations fails to complete because communication is lost between client and server, the server will probably have to take some kind of recovery action such as restoring data to a consistent state and freeing resources.

The stub detects outstanding context when it marshalls context handle parameters. Outstanding context is considered to exist from the point at which a non-**NULL** pointer value is returned, until a **NULL** pointer value is returned. When outstanding context exists, the server stub code will call a context rundown routine in response to certain exceptions that indicate a loss of contact with the client. You should note that the exact timing of the call depends on the transport. In particular, with the connectionless protocol, servers that maintain context for clients expect clients to indicate periodically that they are still running. If the server fails to hear from the client during a specified timeout period, the server will assume that the client has stopped, and will call the context rundown routine. This can mean a substantial delay between the time the client actually fails and the time at which context maintained for the client is actually cleaned up. If the context being held represents a scarce resource on the server, one consequence of the delayed rundown may be that failed calls continue to hold the scarce resource for some time before it is made available again.

Since a context handle may be freely shared among threads of the calling client context, it is possible for outstanding context to exist for more than one call simultaneously. Such shared context is considered to be outstanding as long as it is outstanding for any of the participating threads. Also, any communication failures are likely to be detected at different times for each such call thread, and the difference in timing may be especially noticeable in the case of the connectionless protocol. Context rundown occurs only after all server call threads have been terminated. This means that call operations in progress on the server need not be concerned that the context they are operating on will be changed unexpectedly. Imagine a situation in which context handles represent open file descriptors, and the rundown routine closes the files. A manager thread that shares these descriptors via a context handle is guaranteed that the files will remain open even if a communications failure is detected in another thread that also is using the same context handle.

```
/******
*
*
* store_handle_t_rundown -- Closes the opened storelet.
*
*
******/
/***************************************************************************/
void
store_handle_t_rundown(
        store_handle_t store_h
)
{
        error_status_t st;

        printf("Store Manager: Running down context.\n");
        store_close(&store_h, &st);
}
```

## Binding and Security Information

One element that is clearly missing from the context handle sample code is any access checking. To do this it is necessary to get the client binding, although it may not be immediately obvious how to do this with a context handle. The answer is actually quite simple, but to understand it, it helps to have a clear idea of how binding parameters operate in RPC.

Every call requires binding information, whether this is supplied explicitly as a binding parameter or not. When a call is made with a binding handle, the client uses cached binding information associated with the binding handle. When no binding handle parameter is passed, the client derives the binding information it needs by some other means. For example, with a context handle, the client uses cached binding information associated with the context handle.

Even when an explicit binding handle parameter is present, the handle is not marshalled as call data in the same way other call parameters are. Similarly, on the server side, when a binding handle parameter is present in a manager operation, it is unmarshalled simply as a reference to the binding information cached by the server runtime for the call. It is irrelevant whether the call was made with an explicit binding handle parameter on the client side. Therefore, it is perfectly possible for a server manager operation to have a binding handle as a parameter even when the client RPC call is made without an explicit binding parameter.

The mechanics of this are to use different **.acf** declarations on the client and server sides. The **.idl** file declaration for the operation does not declare an explicit binding handle parameter, but the server **.acf** file applies the **[explicit_handle]** attribute to the operation. This results in a server stub that expects to unmarshall a binding handle as the first parameter of the operation, while the client stub does not expect an explicit binding handle parameter for the call.

An example of a server side **.acf** file for the store interface is as follows:

```
/* store.acf - server side
 * Unmarshal a client binding handle on each call
 */

interface store
{
    store_open();
    [explicit_handle]store_close();
    [explicit_handle]store_set_ptr();
    [explicit_handle]store_read();
    [explicit_handle]store_write();
}
```

You could achieve the same effect by using different **.idl** files for the client and server, but this is not recommended. The **.idl** file serves as the canonical representation of an interface and hence should be the same for all clients and servers.

This technique can be used in a number of ways: for example, to permit the client to use implicit binding while the server manager operations extract authorization information from a client binding handle. In the case of a context handle, the principle is the same. You use the server **.acf** declarations to add a binding parameter to the call on the server side. The client continues to call using the context handle, while the server manager receives the client binding as a first extra parameter. In the case of the sample code, the client calls to the store interface remain the same, but the server manager implementations now contain an extra parameter. For example:

```
    void
    store_write(
        handle_t IDL_handle,
        store_handle_t store_h,
        unsigned32 buf_size,
        store_buf_t buffer,
        unsigned32 *data_size,
        error_status_t *status
    )
    {
        store_spec_t *spec;
        store_hdr_t *hdr;

        if (check_access(IDL_handle, sec_acl_perm_write) == 0)
        {
             *status = str_s_no_perms;
            return;
        }
        .
        .
        .
    }
```

## Pipes

Pipes are a mechanism for efficiently handling large quantities of data by overlapping the transfer and processing of data. Input data is transferred in chunks to the server for processing, and output data is processed by the server in chunks and transferred to the client. A pipe is declared in a type definition of an interface definition and the data type is used as parameters in the operations of the interface. The server manager calls stub pipe support routines in a loop, and the client stub calls pipe support routines that the client application must provide.

One of the pipe support routines that the client must provide is an *alloc* routine, which allocates a buffer for each chunk of pipe data. Given that pipes are intended to process data asynchronously, consuming it as it arrives, the *alloc* routine should not just blindly allocate a new buffer each time it is called, since the net effect would be to allocate space for the whole stream. A reasonable approach is either to declare a buffer statically or allocate it on the first call (per thread), and thereafter simply return the same buffer. The following code example shows the form an *alloc* routine takes in client application code.

```
#define CLIENT_BUFFER_SIZE 2048
idl_byte client_buffer[CLIENT_BUFFER_SIZE];

void client_alloc (state, bsize, buf, bcount)
    rpc_ss_pipe_state_t state;
    unsigned int bsize;
    byte **buf;
    unsigned int *bcount;
{
    *buf = client_buffer;
    *bcount = CLIENT_BUFFER_SIZE;
}
```

## Input Pipes

In the following example, a client sends the contents of a file to a server as a set of chunks allocated from the same static buffer. The chunks are processed (in this case, simply printed) as they arrive.

The declaration in the interface definition is as follows:

```
typedef pipe char test_pipe_t;

void pipe_test1(
    [in] handle_t handle,
    [in] test_pipe_t test_pipe,
    [out] error_status_t *status
);
```

Note that the pipe is declared as a **typedef**, which results in an IDL-generated C typedef for **test_pipe_t**: a structure containing pointers to the pipe support routines and a pipe state field. The server manager and client code then implement the pipe in a complementary fashion.

For an **[in]** pipe, the server manager code consists of a cycle of calls to the **test_pipe.pull** routine (a server stub routine) which terminates when a zero-length chunk is received:

```
    void
    pipe_test1(
        handle_t binding_h,
        test_pipe_t test_pipe,
        error_status_t *status
    )
    {
        char buffer[SBUFFSIZE];
        int count;
        char *cptr;
        do
        {
            (*(test_pipe.pull))(test_pipe.state, buffer, SBUFFSIZE, &count);
            for (cptr = buffer; cptr < buffer + count; cptr++)
                putchar(*cptr);
        } while (count > 0);
    }
```

Using the buffer supplied by the manager, the **test_pipe.pull** routine unmarshalls an amount of data that is nonzero, but not more than the buffer can hold. There is no guarantee that the buffer will be filled. The actual amount of data in the buffer is indicated by the **count** parameter returned in the **test_pipe.pull** routine. This count equals the number of **test_pipe_t** data elements in the buffer.

The **test_pipe.pull** routine signals the end of data in the pipe by returning a chunk whose count is 0 (zero). Any attempt to pull data from the pipe after the zero-length chunk has been encountered will cause an exception to be raised. The **in** pipes must be processed in the order in which they occur in the operation signature. Attempting to pull data from an **in** pipe before End-of-Data on any preceding **in** pipe has been encountered will result in an exception being raised. If the manager code attempts to write to an **out** pipe or return control to the server stub before End-of-Data has been encountered on the last **in** pipe, an exception will be raised. (Note that there is no guarantee that chunks seen by the manager will match the chunks supplied by the client's *pull* routine.)

The client application code must supply *pull* and *alloc* routines and a pipe state. These routines must work together to produce a sequence of pointers to chunks, of which only the last is empty. In the following example, the client code provides a **test_pipe.pull** routine that reads chunks of the input file into a buffer and returns a count of the chunk size, returning a zero count when the end of the file is reached. The pipe state block is used here simply as a convenient way to make the file state available to the pull routine. Applications need not make any use of the pipe state.

```
    /* Client declares types and routines */

    typedef struct client_pipe_state_t {
        idl_char *filename;
        idl_boolean file_open;
        int file_handle;
    } client_pipe_state_t;

    client_pipe_state_t client_in_pipe_state = {false, 0};
```

```
void client_pull(state,buf,esize,ecount)
    client_pipe_state_t * state;
    byte *buf;
    unsigned int esize;
    unsigned int *ecount;
{
    if ( ! state->file_open )
    {
        state->file_handle = open(state->filename,O_RDONLY);
        if (state->file_handle == -1)
        {
            printf("Client couldn't open %s\n", state->filename);
            exit(0);
        }
        state->file_open = true;
    }
    *ecount = read( state->file_handle, buf, esize );
    if (*ecount == 0)
    {
        close(state->file_handle);
        state->file_open = false;
    }
}
```

Finally, the client must do the following:

1. Allocate the **test_pipe_t** structure.
2. Initialize the **test_pipe_t.pull**, **test_pipe_t.alloc**, and **test_pipe_t.state** fields.
3. Include code where appropriate for checking the **pipe_t.state** field.
4. Pass the structure as the pipe parameter. The structure can be passed either by value or by reference, as indicated by the signature of the operation that contains the pipe parameter.

```
/* Client initializes pipe */
test_pipe_t test_pipe;

test_pipe.pull = client_pull;
test_pipe.alloc = client_alloc;
test_pipe.state = (rpc_ss_pipe_state_t)&client_in_pipe_state;

/* Client makes call */

pipe_test1(binding_h, test_pipe, &status);
```

To transmit a large amount of data that is already in the proper form in memory (that is, the data is already an array of **test_pipe_t**), the client application code can have the *alloc* routine allocate a buffer that already has the information in it. In this case, the *pull* routine becomes a null routine.

## Output Pipes

An **[out]** pipe is implemented in a similar way to an input pipe, except that the client and server make use of the *push* routine instead of the *pull* routine. The following samples show an **[out]** pipe used to read the output from a shell command executed by the server.

The declarations in the interface definition are as follows:

```
typedef pipe char test_pipe_t;

void pipe_test2(
    [in] handle_t handle,
    [in, string]  char cmd[],
    [out] test_pipe_t *test_pipe,
    [out] error_status_t *status
);
```

The server manager routines demonstrate a couple of possible implementations. In each case, the manager makes a cycle of calls to the server stub's push routine, ending by pushing a zero-length chunk:

```
#include <dirent.h>
#define SBUFFSIZE 256

void
pipe_test2(
    handle_t binding_h,
    idl_char *cmd,
    test_pipe_t *test_pipe,
    error_status_t *status
)
{

    DIR *dir_ptr;
    struct dirent *directory;

    char buffer[SBUFFSIZE];
    FILE *str_ptr;
    int n;

    /* An elementary mechanism to execute a command and get the
     * output back. Note that popen() and fread() are thread-safe,
     * so the whole process won't block while the call thread waits
     * for them to return.
     *
     * This is potentially a dangerous operation!
     * Here we'll only allow a couple of "safe" commands.
     */

    if (!strcmp(cmd, "ps") || !strcmp(cmd, "ls"))
    {
        if ((str_ptr = popen(cmd, "r")) == NULL)
            return;
        while ((n = fread(buffer, sizeof(char), SBUFFSIZE, str_ptr)) > 0)
        {
            (*(test_pipe->push))(test_pipe->state, buffer, n);
        }
        (*(test_pipe->push))(test_pipe->state, buffer, 0);
        fclose(str_ptr);
    }

    /* Here's another method: list an arbitrary directory
     * This time, we buffer the directory names as null-terminated
     * strings of various lengths. The client will need to provide
     * formatting of the output stream, for example, by substituting
     * a CR for each NULL byte.
     */

    /*
    if ((dir_ptr = opendir(cmd)) == NULL)
    {
        printf("Can't open directory %s\n", cmd);
        return;
    }
    while ((directory = readdir(dir_ptr)) != NULL)
    {
        if (directory->d_ino == 0)
            continue;
        (*(test_pipe->push))(test_pipe->state, directory->d_name,
                             strlen(directory->d_name)+1);
    }
    (*(test_pipe->push))(test_pipe->state, directory->d_name, 0);
    closedir(dir_ptr);

    */

    *status = error_status_ok;
}
```

The stub enforces well-behaved pipe filling by the manager by raising exceptions as necessary. After all **in** pipes have been drained completely, the **out** pipes must be completely filled, in order.

The client code uses the same declarations as in the input pipe example, except that instead of using a **client_pull** routine it uses a **test_push** routine that prints out the contents of each received buffer:

```
    /*
     * Our push routine prints each received buffer-full.
     */

    void test_push(
        rpc_ss_pipe_state_t *state,
        idl_char *buf,
        unsigned32 count
    )
    {
        unsigned_char_t *cptr;
        for (cptr = buf; cptr < buf + count; cptr++)
        {
            /* For the second, directory reading example, uncomment the
               following:
            if (*cptr == 0)
                *cptr = '\n';
             */
            putchar(*cptr);
        }
    }
```

For an **out** pipe, the client code must do the following:

1. Allocate the **test_pipe_t** structure.
2. Initialize the **test_pipe.push** and **test_pipe.state** fields.
3. Pass the structure as the pipe parameter, either by value or by reference.

```
    test_pipe_t test_pipe;

    test_pipe.alloc = (void (*)())client_alloc;
    test_pipe.push = (void (*)())test_push;
    test_pipe.state = (rpc_ss_pipe_state_t)&out_test_pipe_state;

    pipe_test2(binding_h, cmd, &test_pipe, &status);
```

The client stub unmarshalls chunks of the pipe into a buffer and calls back to the application, passing a reference to the buffer. To allow the application code to manage its memory usage, and possibly avoid unnecessary copying, the client stub first calls back to the application's **test_pipe.alloc** routine to get a buffer. In some cases, this may result in the **test_pipe.push** routine's not having any work to do.

The client stub may go through more than one (**test_pipe.alloc**, **test_pipe.push**) cycle in order to unmarshall data that the server marshalled as a single chunk. Note that there is no guarantee that chunks seen by the client stub will match the chunks supplied by the server's push routine.

## Pipe Summary

The pipe examples show how the client and server tasks are complementary. The client implements the appropriate callback routines (**test_pipe.alloc** and either **test_pipe.push** or **test_pipe.pull**), and the server manager makes a cycle of calls to either **test_pipe.push** or **test_pipe.pull** of the stub. The application code gives the illusion that the server manager is calling the client-supplied callbacks. In fact, the manager is actually calling stub-supplied callbacks, and the client callbacks are asynchronous: a server manager call to one of the callback routines does not necessarily result in a call to the corresponding client callback.

One result of this is that the client and server should not count on the chunk sizes being the same at each end. For example, in the last directory reading example, the manager calls the **test_pipe.push** routine once with each **NULL**-terminated file name. However, the client **test_push** routine does not necessarily receive the data stream one file name at a time. For example, if the **test_push** routine attempted to print the file names using `printf("%s\n",buf);`, it might fail. An interesting exercise would be to add **printf()**s to the client callbacks and the server manager to show when each callback is made.

Note also that the use of the pipe *state* field by the client is purely local and entirely at the discretion of the client. The state is not marshalled between client and server, and the server stubs use the local *state* field in a private manner. The server manager should not alter the state field.

Pipes may also be **[in,out]**, although the utility of this construct is somewhat limited. Ideally, a client would like to be able to pass a stream of data to the server and have it processed and returned asynchronously. In practice, the input and output streams must be processed synchronously: all input processing must be finished before any output processing can be done. This means that **[in, out]** pipes, while they can reduce latency within both the server and the client, cannot reduce latency between server and client: the client must still wait for all server processing to finish before it can begin to process the returned data stream.

For an **in,out** pipe, both the **pull** routine (for the **in** direction) and a **push** routine (for the **out** direction) must be initialized, as well as the **alloc** routine and the state. During the last **pull** call (when it will return a zero count to indicate that the pipe is drained), the application's **pull** routine must reinitialize the pipe state so that the pipe can be used by the **push** routine correctly.

## Nested Calls and Callbacks

A called remote procedure can call another remote procedure. The call to the second procedure is nested within the first; that is, the second call is a nested remote procedure call. A nested call involves the following general phases (as illustrated in Figure 40 on page 198):

**1** A client makes an initial RPC to the first remote procedure.

**2** The first remote procedure makes a nested call to the second remote procedure.

**3** The second remote procedure runs the nested call and returns it to the first remote procedure.

**4** The first remote procedure then resumes running the initial call.

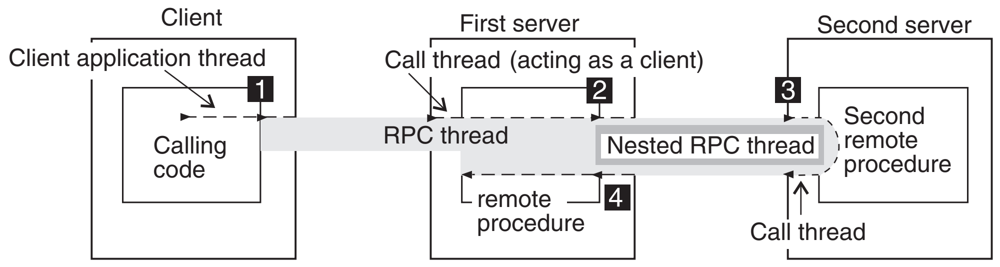


*Figure 40. Phases of a Nested RPC*

A specialized form of a nested RPC shown in Figure 41, involves a called remote procedure that is making an RPC to the address space of the calling client application thread. Calling the client's address space requires that a server application thread be listening in that address space. Also, the second remote procedure needs a server binding handle for the address space of the calling client.

The remote procedure can convert the client binding handle into a server binding handle by calling the **rpc_binding_server_from_client** routine. This routine returns a partially bound binding handle. (The server binding information lacks an endpoint.) For a nested RPC to find the address space of the calling client, the server must ensure that the partially bound binding handle is filled in with the endpoint of the client address space. Information about the **rpc_binding_server_from_client** routine in the *z/OS DCE Application Development Reference* discusses alternatives for ensuring that the endpoint is obtainable for a nested RPC.

Using the server binding handle, a remote procedure can attempt a nested RPC. The nested call involves the general phases illustrated in Figure 41.

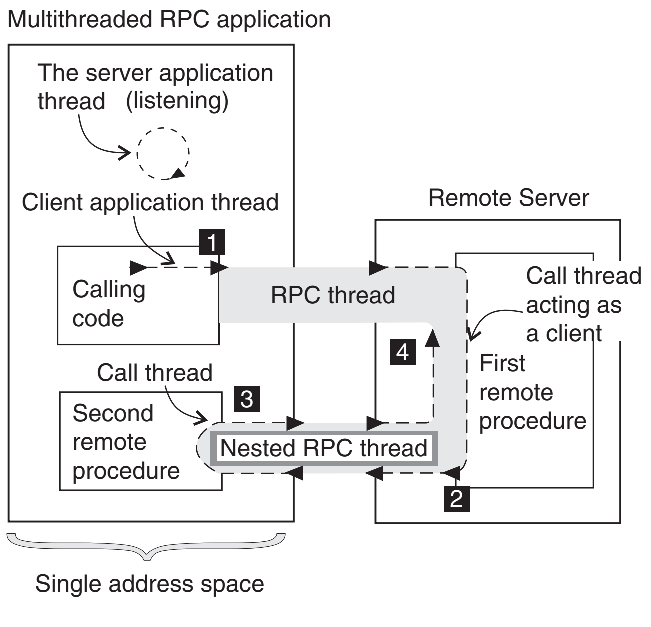


*Figure 41. Phases of a Nested RPC to a Client Address Space*

The application threads in Figure 41 perform the following activities indicated by the reference keys.

**1** A client application thread from a multithreaded RPC application makes an initial RPC call to the first remote procedure.

2 After converting the client binding handle into a server binding handle and obtaining the endpoint for the address space of the calling client application thread, the first remote procedure makes a nested call to the second remote procedure at that address space.

3 The second remote procedure runs the nested call and returns it to the first remote procedure.

4 The first remote procedure then resumes running the initial call.

## Routing Remote Procedure Calls

The following subsections discuss routing incoming RPCs between their arrival at a server's system and the server's start up of the requested remote procedure. The routing steps are:

1 If a client has a partially bound server binding handle, before sending a call request to a server, the client runtime must get the endpoint of a compatible server from the endpoint map service of the server's system. This endpoint becomes the server address for a call request.

2 When the request arrives at the endpoint, the server's system places it in a request buffer belonging to the corresponding server.

3 As one of its scheduled tasks, the server gets the incoming calls from the request buffer. The server either accepts or rejects an incoming call, depending on available resources. If no call thread is available, an accepted call is queued to wait its turn for an available call thread.

4 The server then allocates an available call thread to the call.

5 The server identifies the appropriate manager for the called remote procedure and runs the procedure in that manager to run the call.

6 When the call thread finishes running a call, the server returns the call's output arguments and control to the client.

Figure 42 on page 200 illustrates these steps.

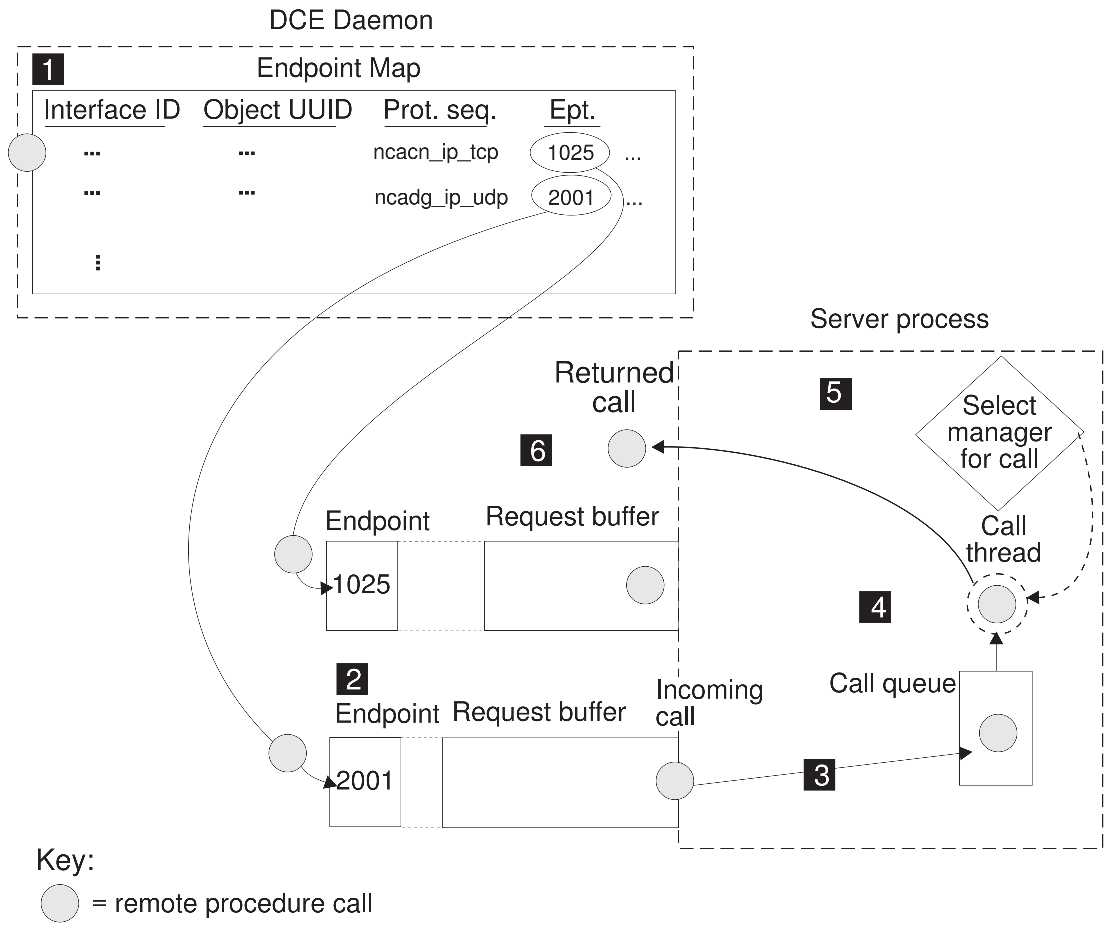


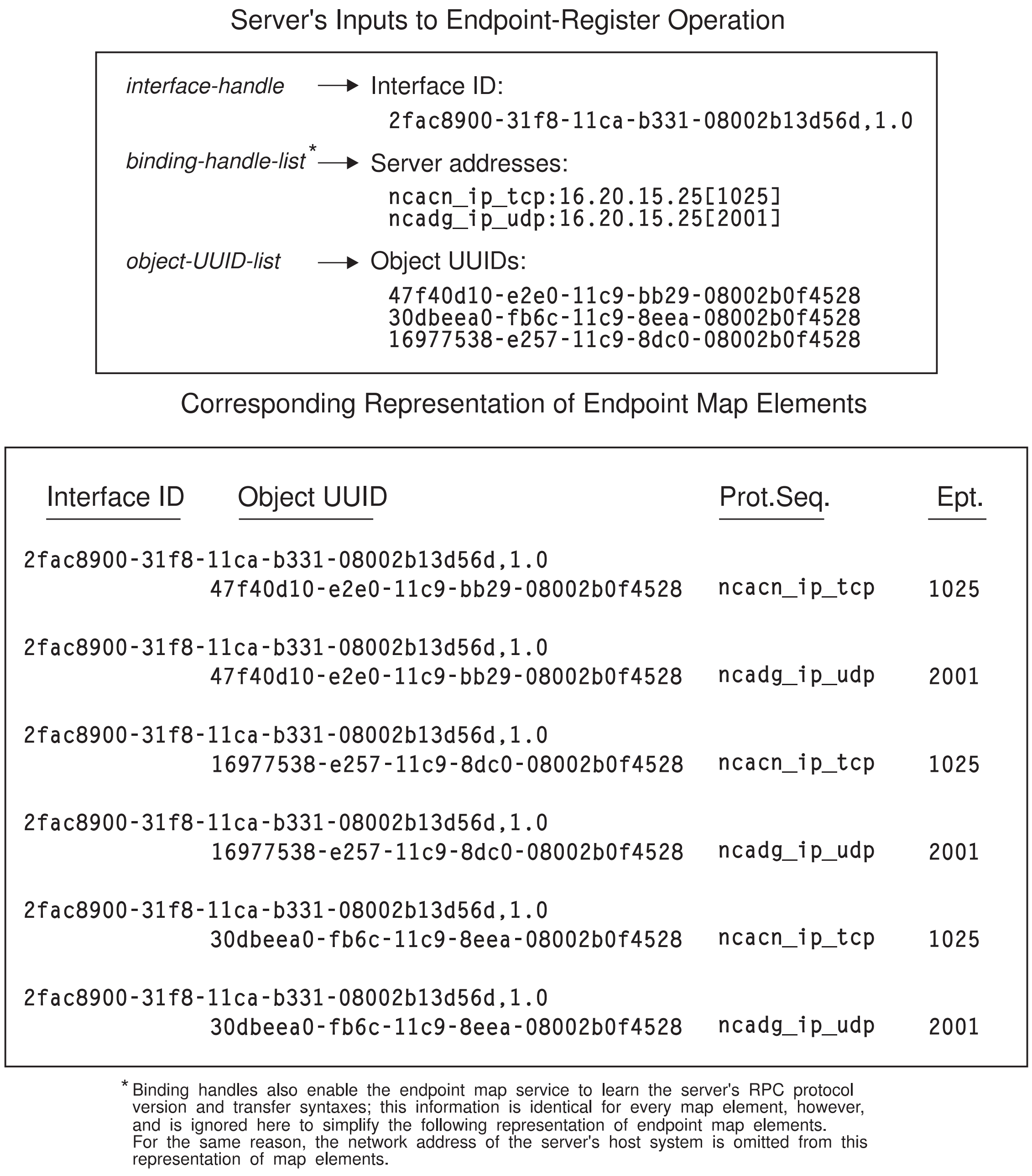
*Figure 42. Steps in Routing Remote Procedure Calls*

The concepts in the following subsections are for the advanced RPC developer. "Obtaining an Endpoint" discusses how clients obtain endpoints when using partially bound binding handles. "Buffering Call Requests" on page 205 and "Queuing Incoming Calls" on page 206 discuss how a system buffers call requests and how a server queues incoming calls; this information is relevant mainly to advanced RPC developers. "Selecting a Manager" on page 209 discusses how a server selects the manager to run a call; it is relevant for developing an application that carries out an interface for different types of RPC objects.

## Obtaining an Endpoint

The endpoint mapper service of **dced** maintains the local endpoint map. The endpoint map is composed of elements. Each map element contains fully bound server binding information for a potential binding and an associated interface identifier and object UUID, which may be nil. Optionally, a map element can also contain an annotation such as the interface name.

Servers use the local endpoint map service to register their binding information. Each interface for which a server must register binding information requires a separate call to an **rpc_ep_register...()** routine, which calls the endpoint map service. The endpoint map service uses a new map element for every combination of binding information specified by the server. Figure 43 on page 201 shows the correspondence between server binding information specified by a server and a graphic representation of the resulting endpoint map elements.



Server's Inputs to Endpoint-Register Operation

*interface-handle* → Interface ID:

```
2fac8900-31f8-11ca-b331-08002b13d56d,1.0
```

*binding-handle-list** → Server addresses:

```
ncacn_ip_tcp:16.20.15.25[1025]
ncadg_ip_udp:16.20.15.25[2001]
```

*object-UUID-list* → Object UUIDs:

```
47f40d10-e2e0-11c9-bb29-08002b0f4528
30dbeea0-fb6c-11c9-8eea-08002b0f4528
16977538-e257-11c9-8dc0-08002b0f4528
```

Corresponding Representation of Endpoint Map Elements

| Interface ID | Object UUID | Prot.Seq. | Ept. |
|---|---|---|---|
| 2fac8900-31f8-11ca-b331-08002b13d56d,1.0 | 47f40d10-e2e0-11c9-bb29-08002b0f4528 | ncacn_ip_tcp | 1025 |
| 2fac8900-31f8-11ca-b331-08002b13d56d,1.0 | 47f40d10-e2e0-11c9-bb29-08002b0f4528 | ncadg_ip_udp | 2001 |
| 2fac8900-31f8-11ca-b331-08002b13d56d,1.0 | 16977538-e257-11c9-8dc0-08002b0f4528 | ncacn_ip_tcp | 1025 |
| 2fac8900-31f8-11ca-b331-08002b13d56d,1.0 | 16977538-e257-11c9-8dc0-08002b0f4528 | ncadg_ip_udp | 2001 |
| 2fac8900-31f8-11ca-b331-08002b13d56d,1.0 | 30dbeea0-fb6c-11c9-8eea-08002b0f4528 | ncacn_ip_tcp | 1025 |
| 2fac8900-31f8-11ca-b331-08002b13d56d,1.0 | 30dbeea0-fb6c-11c9-8eea-08002b0f4528 | ncadg_ip_udp | 2001 |

* Binding handles also enable the endpoint map service to learn the server's RPC protocol version and transfer syntaxes; this information is identical for every map element, however, and is ignored here to simplify the following representation of endpoint map elements. For the same reason, the network address of the server's host system is omitted from this representation of map elements.

*Figure 43. Mapping Information and Corresponding Endpoint Map Elements*

An RPC made with server binding information that lacks an endpoint uses an endpoint from the endpoint map service. This endpoint must come from binding information of a compatible server. The map element of a compatible server contains the following:

- A compatible interface identifier

  The requested interface UUID and compatible version numbers are necessary. For the version to be compatible, the major version number requested by the client and registered by the server must be identical, and the requested minor version number must be less than or equal to the registered minor version number.

- The requested object UUID, if registered for the interface

- A server binding handle that refers to compatible binding information that contains:

- A protocol sequence from the client’s server binding information
- The same RPC protocol major version number that the client runtime supports
- At least one transfer syntax that matches one used by the client’s system

To identify the endpoint of a compatible server, the endpoint service uses the following rules:

1. If the client requests a non-nil object UUID, the endpoint map service begins by looking for a map element that contains both the requested interface UUID and object UUID.
   a. On finding an element containing both of the UUIDs, the endpoint map service selects the endpoint from that element for the server binding information used by the client.
   b. If no element contains both UUIDs, the endpoint map service discards the object UUID and starts over (see rule 2).
2. If the client requests the nil object UUID (or if the requested non-nil object UUID is not registered), the endpoint map service looks for an element containing the requested interface UUID and the nil object UUID.
   a. On finding that element, the endpoint map service selects the endpoint from the element for the client’s server binding information.
   b. If no such element exists, the lookup fails.

The RPC protocol service inserts the endpoint of the compatible server into the client’s server binding information.

The flow chart in Figure 44 on page 203 illustrates the decisions the endpoint map service makes when looking up an endpoint for a client.

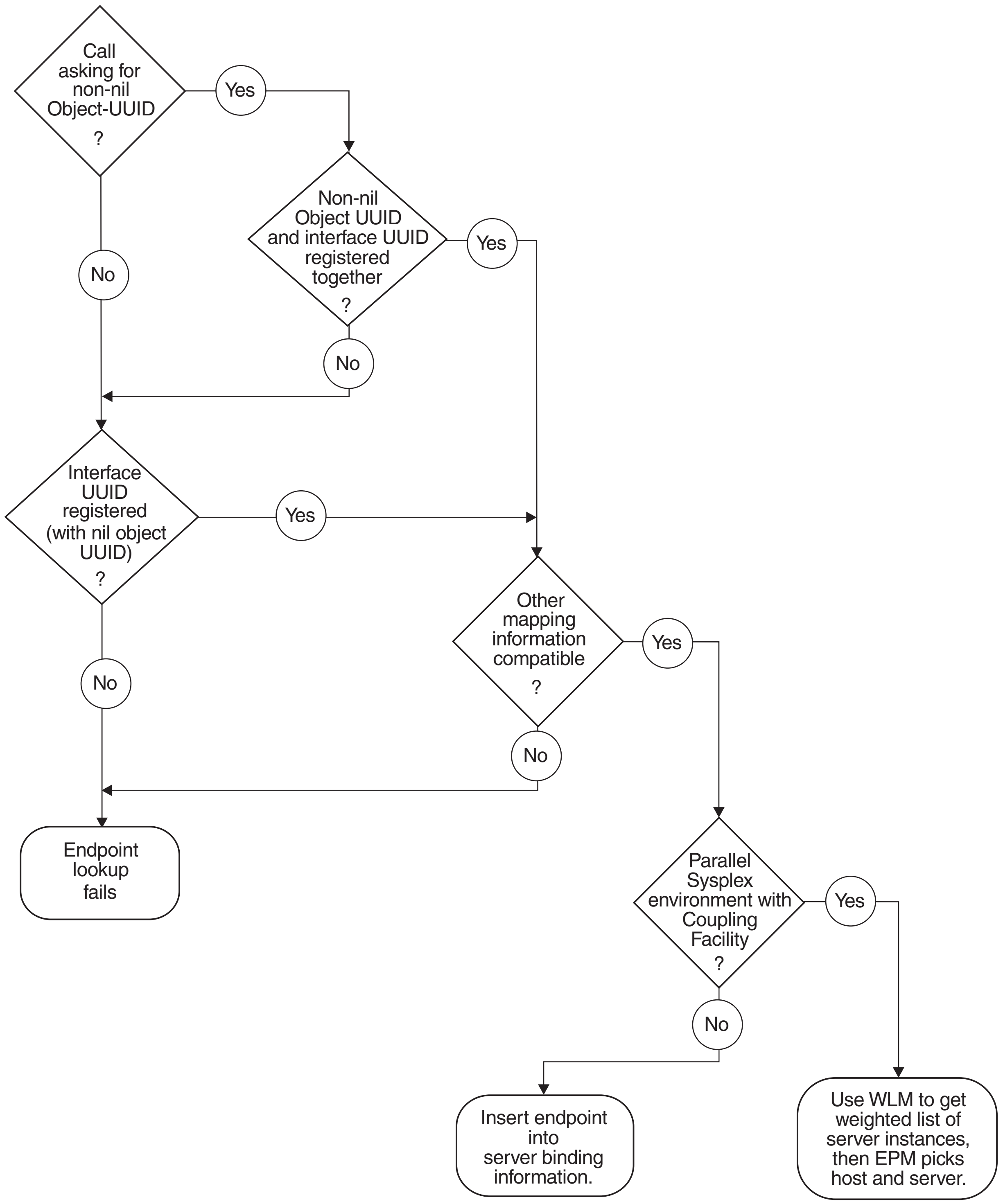

Call asking for non-nil Object-UUID ?
Yes
No
Non-nil Object UUID and interface UUID registered together ?
Yes
No
Interface UUID registered (with nil object UUID) ?
Yes
No
Other mapping information compatible ?
Yes
No
Endpoint lookup fails
Parallel Sysplex environment with Coupling Facility ?
Yes
No
Insert endpoint into server binding information.
Use WLM to get weighted list of server instances, then EPM picks host and server.

*Figure 44. Decisions for Looking Up an Endpoint*

You can design a server to allow the coexistence on a host system of multiple interchangeable instances of a server. Interchangeable server instances are identical, except for their endpoints. That is, they offer the same RPC interfaces and objects over the same network (host) address and protocol sequence pairs. For clients, identical server instances are fully interchangeable.

Having identical server instances is particularly useful in a Parallel Sysplex® environment. If you also have the Coupling Facility (XCF) hardware feature and the Workload Manager (WLM) software, you can balance workloads among different hosts (or within one host) having identical server instances.

Usually, for each such combination of mapping information (that is, each identical pair of interface and object), the endpoint map service stores only one endpoint at a time. When a server registers a new endpoint for mapping information that is already registered, the endpoint map service replaces the old map element with the new one.

For interchangeable server instances to register their endpoints in the local endpoint map, they must instruct the endpoint map service not to replace any existing elements for the same interface identifier and object UUID. Each server instance can create new map elements for itself by calling the **rpc_ep_register_no_replace()** routine, or, in a Parallel Sysplex environment with workload balancing, the **rpc_ep_register_no_replace_wlb()** routine.

When a client uses a partially bound binding handle, load sharing among interchangeable server instances depends on the RPC protocol the client is using.

- **Connectionless (datagram) protocol**
  - In most cases:

    The map service selects the first map element with compatible server binding information. If necessary, a client can achieve a random selection among all the map elements with compatible binding information. However, this requires that before making an RPC, the client needs to resolve the binding by calling the **rpc_ep_resolve_binding()** routine.
  - In a Parallel Sysplex environment:

    For registered endpoints with the *activated* variable in the **true** state, the endpoint map service calls WLM, which returns a list of identical server instances that are weighted according to current workload. The map service then uses the weights to select a server instance.
- **Connection-oriented protocol**
  - In most cases:

    The client RPC runtime uses the **rpc_ep_resolve_binding()** routine, and the endpoint map service selects randomly among all the map elements of compatible servers.
  - In a Parallel Sysplex environment:

    The client RPC runtime calls the endpoint map service using **rpc_ep_resolve_binding()**. For registered endpoints with the *activated* variable in the **true** state, the map service calls WLM, which returns a list of identical server instances that are weighted according to current workload. The map service then uses the weights to select a server instance.

For an alternative selection criterion, a client can call the **rpc_mgmt_ep_elt_inq_**{**begin**,**next**,**done**}() routines and use an application-specific routine to select from among the binding handles returned to the client.

When a server stops running, its map elements become outdated. Although the endpoint map service routinely removes any map element containing an outdated endpoint, a lag time exists when stale entries remain. If a remote procedure call uses an endpoint from an outdated map element, the call fails to find a server. To avoid clients getting stale data from the endpoint map, before a server stops, it should remove its own map elements.

A server also has the option of removing any of its own elements from the local endpoint map and continuing to run. In this case, an unregistered endpoint remains accessible to clients that know it.

## Buffering Call Requests

Call requests for RPC servers come into the RPC runtime over the network. For each endpoint that a server registers (for a given protocol sequence), the runtime sets up a separate request buffer. A request buffer is a first-in, first-out queue where an RPC system temporarily stores call requests that arrive at an endpoint of an RPC server. The request buffers allow the runtime to continue to accept requests during heavy activity. However, a request buffer may fill up temporarily, causing the system to reject incoming requests until the server fetches the next request from the buffer. In this case, the calling client can try again, with the same server or a different server. The client does not know why the call is rejected, nor does the client know when a server is available again.

Each server process regularly dequeues requests, one by one, from all of its request buffers. At this point, the server process recognizes them as incoming calls. The interval for removing requests from the buffers depends on the activities of the system and of the server process.

How the runtime handles a given request depends partly on the communications protocol over which it arrives:

- A call over a connectionless transport is routed by the server's system to the call request buffer for the endpoint specified in the call.
- A call over a connection-oriented transport may be routed by the server's system to a request buffer, or the call may go directly to the server process.

  Whether an RPC goes to the request buffer depends on whether the client sends the call over an established connection. If a client makes an RPC without an established connection, the server's system treats the call request as a connection request and places it into a request buffer. If an established connection is available, the client uses it for the RPC. The system handles the call as an incoming call and sends it directly to the server process that owns the connection.

Whether a server gets an incoming call from a request buffer or over an existing connection, the server process manages the call identically. A server process applies a clear set of call-routing criteria to decide whether to send a call immediately, queue it, or reject it (if the server is extremely busy). These call-routing criteria are discussed in "Queuing Incoming Calls" on page 206.

When telling the RPC runtime to use a protocol sequence, a server specifies the number of calls it can buffer for the specified communications protocol (at a given endpoint). Usually, it is best for a server to specify a default buffer size, represented by a literal whose underlying value depends on the communications protocol. The default equals the capacity of a single socket used for the protocol by the server's system.

The default usually is adequate to allow the RPC runtime to accept all the incoming call requests. For a well-known endpoint, the size of a request buffer cannot exceed the capacity of a single socket descriptor (the default size); specifying a higher number causes a runtime error. For well-known endpoints, specify the default for the maximum number of call requests.

For example, consider the request buffer at full capacity as represented in Figure 45 on page 206. This buffer has the capacity to store five requests. In this example, the buffer is full, and the runtime rejects incoming requests, as is happening to the sixth request.

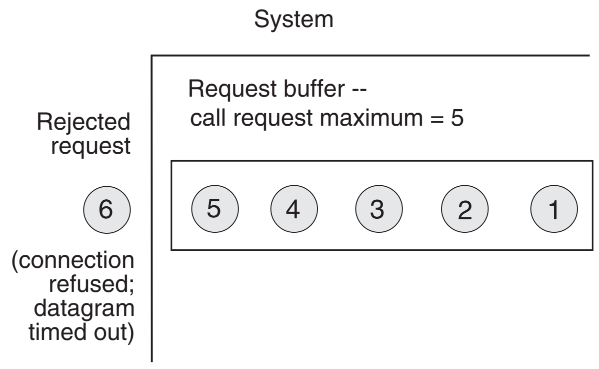


*Figure 45. A Request Buffer at Full Capacity*

## Queuing Incoming Calls

Each server process uses a first-in, first-out call queue. When the server is already running its maximum number of concurrent calls, it uses the queue to hold incoming calls. The capacity of queues for incoming calls is implementation dependent; most implementations offer a small queue capacity, which may be a multiple of the maximum number of concurrently running calls.

A call is rejected if the call queue is full. The appearance of the rejected call depends on the RPC protocol the call is using, as follows:

- **Connectionless (datagram) protocol**

  The server does not notify the client about this failure. The call fails as if the server does not exist, returning an **rpc_s_comm_failure** communications status code (**rpc_x_comm_failure** exception).

- **Connection-oriented protocol**

  The server rejects the call with an **rpc_s_server_too_busy** communications status code (**rpc_x_server_too_busy** exception).

The server process routes each incoming call as it arrives. Call routing is illustrated by the server in Figure 46. This server has the capacity to run only one call concurrently. Its call queue has a capacity of eight calls. This figure consists of four stages (A through D) of call routing by a server process. On receiving any incoming call, the server begins by looking at the call queue.

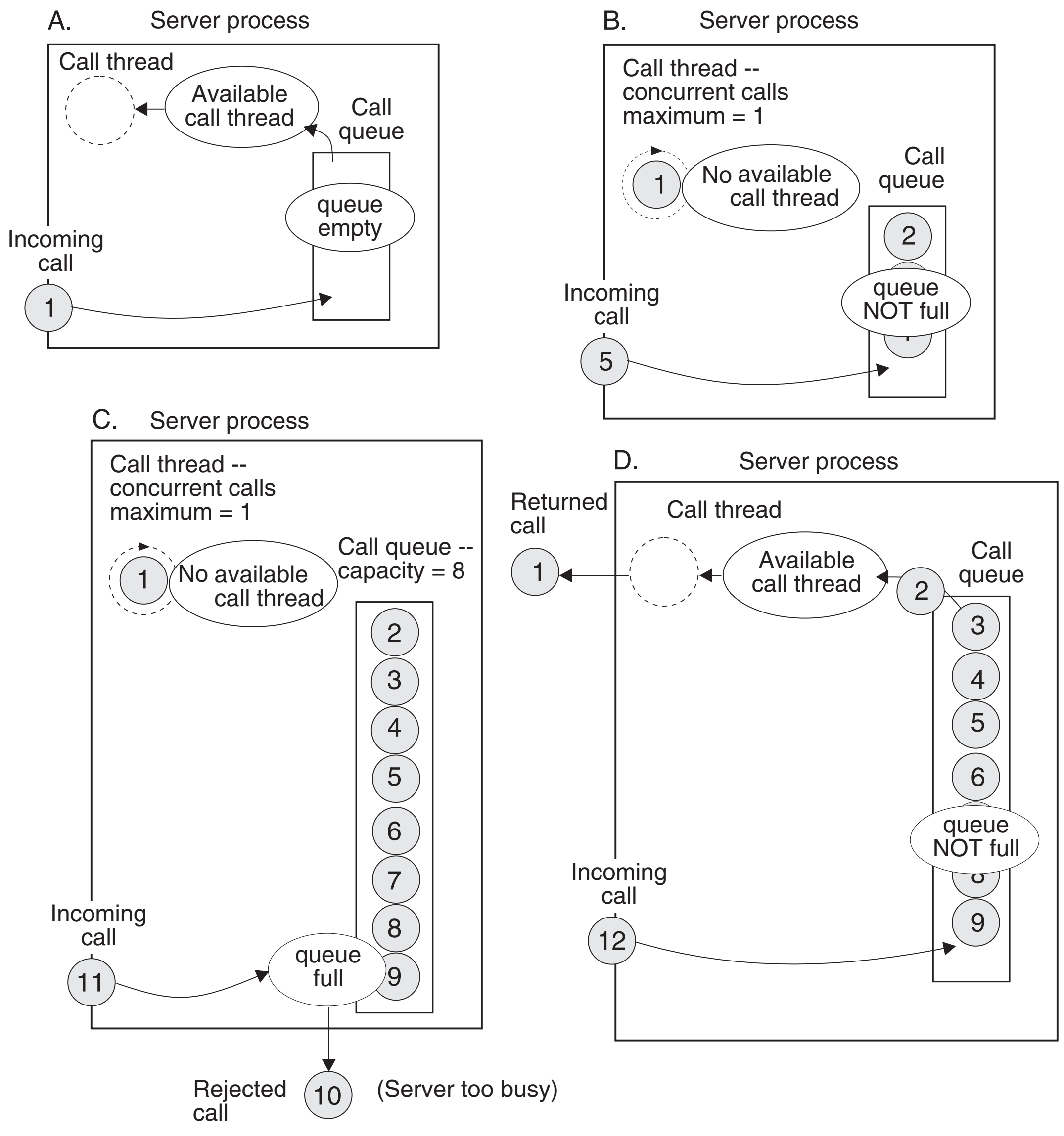


*Figure 46. Stages of Call Routing by a Server Process*

1. In stage A, call **(1)** arrives at a server that lacks any other calls. When the call arrives, the queue is empty and a call thread is available. The server accepts the call and immediately passes it to a call thread. The requested remote procedure runs the call in that thread, which becomes temporarily unavailable.

2. In stage B, call **(5)** arrives. The call queue is partially full, and the server accepts the call and adds it to the end of the queue.
3. In stage C, call **(11)** arrives. The queue is full, so the server rejects this call, as it rejected the previous call, **(10)**. (The caller can try again with the same or a different server.)
4. In stage D, the called procedure has completed the call **(1)**, making the call thread available. The server has removed call **(2)** from the queue and is passing it to the call thread to be run. Thus, the queue is partially empty as call **(12)** arrives, so the server accepts the call and adds it to the queue.

## Dynamic Executor Threads

To improve scalability and increase the performance of the RPC runtime, z/OS DCE dynamically allocates and deallocates system resources such as the number of executor threads required by the server application. A static allocation of executor threads can cause the following problems:

1. Long server start up time resulting from the number of machine instructions required to run the **pthread_create()** API.
2. Waste of critical system resources if a long running server is not busy.
3. The number of servers supported may be limited if all servers specify a large number of executor threads.

In addition, a large number of executor threads can affect overall system performance. Note that all executor threads are heavy-weight threads. When you call **rpc_mgmt_stop_server_listening()** to stop your server, the Task Control Block for this thread is detached.

### Initializing Executor Threads:

This section describes how the number of initial executor threads are created, before the RPC runtime begins to listen for a call.

The number of initial executor threads is calculated based on the number of maximum concurrent calls you specify in the **rpc_server_listen()** API. If you specify **rpc_c_listen_max_calls_default** in the **rpc_server_listen()** call, z/OS DCE sets the maximum number of concurrent calls at 50 % of the process thread limit for the process. You can change the process thread limit by setting both the **MAXTHREADS** and **MAXTHREADTASKS** parameters in the BPXPRMxx parmlib member of the SYS1.PARMLIB data set to the required value. A value of 500 or greater is recommended for DCE applications. The parameters contained in BPXPRMxx control the UNIX System Services environment, the hierarchical file system, and the sockets file systems. The system uses these parameter values to initialize the UNIX System Services kernel. For more information on the above, refer to *z/OS C/C++ Run-Time Library Reference*, SA22-7821.

To calculate the number of initial executor threads, use the following formulas:

| **Number of Concurrent Calls (x)** | **Number of Initial Threads (y)** |
|---|---|
| $x \le 10$ | $y = x$ |
| $10 < x \le 50$ | $y = ((x - 10) \times 0.25) + 10$ |
| $x > 50$ | $y = ((x - 50) \times 0.20) + 10$ |

**Increasing Executor Threads:** The z/OS DCE product increases the executor thread pool under the following conditions:

- The response to the current RPC has been sent, and the current number of idling executor threads is less than the minimum threshold value which is set to the initial value.
- An RPC has arrived, but no idle executor thread is available, and the total number of executor threads created is less than the number of concurrent calls specified in the **rpc_server_listen()** API.

In the first case, after an incoming call is processed, the RPC runtime checks the current number of idle executor threads and decides whether to create, destroy, or reuse the executor thread. If the number is less than the minimum threshold, the RPC runtime increases the number of idle executor threads. The number of newly created executor threads depends on the difference between the current number of idle executor threads and the minimum threshold value. The DCE RPC runtime increases the number of executor threads by twenty percent of this difference. This percentage dampens the number of executor threads created to avoid creating too many executor threads. For example, if the minimum threshold value is 20 and the current number of idle threads is 15, only one executor thread is created. If the number of idle threads is greater than or equal to the maximum threshold, the executor thread may be destroyed as described in “Decreasing Executor Threads.”

In the second case, the RPC runtime creates a new executor thread, unless the maximum number of executor threads that have been created reaches the maximum allowed.

More information on the creation of executor threads can be found in *z/OS DCE Application Development Reference* in the description of the **rpc_server_listen()** API.

**Decreasing Executor Threads:** After an incoming call is processed, the RPC runtime checks the current number of idle executor threads. If it is greater than or equal to the maximum threshold value, the RPC runtime destroys the current executor thread.

**Disabling Dynamic Executor Threads:** To improve performance and scalability of RPC, the z/OS DCE default setting for the executor thread pool is to enable dynamic executor threads. If you want to disable the dynamic executor threads and have the RPC runtime use static executor threads, set the environment variable **_EUV_RPC_DYNAMIC_POOL** to 0. If this environment variable is not defined, the default is to use the dynamic threads.

## Selecting a Manager

Unless an RPC interface is used for more than one specific type of object, selecting a manager for an incoming call is a simple process. When registering an interface with a single manager, the server specifies the nil type UUID for the manager type.[2] In the absence of any other manager, all calls, regardless of whether they request an object, go to the nil type manager.

The situation is more complex when a server registers multiple managers for an interface. The server runtime must select from among the managers for each incoming call to the interface. DCE RPC requires a server to set a non-nil type UUID for a set of objects, and for any interface that will access the objects, to register a manager with the same type UUID.

To send an incoming call to a manager, a server does the following:

1. If the call contains the nil object UUID, the server looks for a manager registered with the nil type UUID (the nil type manager).

[2] The API uses **NULL** to specify a synonym to the address of the nil UUID, which contains only zeros.

- If the nil type manager exists for the requested interface, the server sends the call to that manager.
- Otherwise, the server rejects the call.

2. If the call contains a non-nil object UUID, the server looks to see whether it has set a type for the object (by assigning a non-nil type UUID).

   If the object lacks a type, the server looks for the nil type manager.

   - If the nil type manager exists for the requested interface, the server sends the call to that manager.
   - Otherwise, the server rejects the call.

3. If the object has a type, the call requires a remote procedure of a manager whose type matches the object's type. In its absence, the RPC runtime rejects the call.

The flow chart in Figure 47 on page 211 illustrates the decisions a server makes to select a manager to which to send an incoming call.

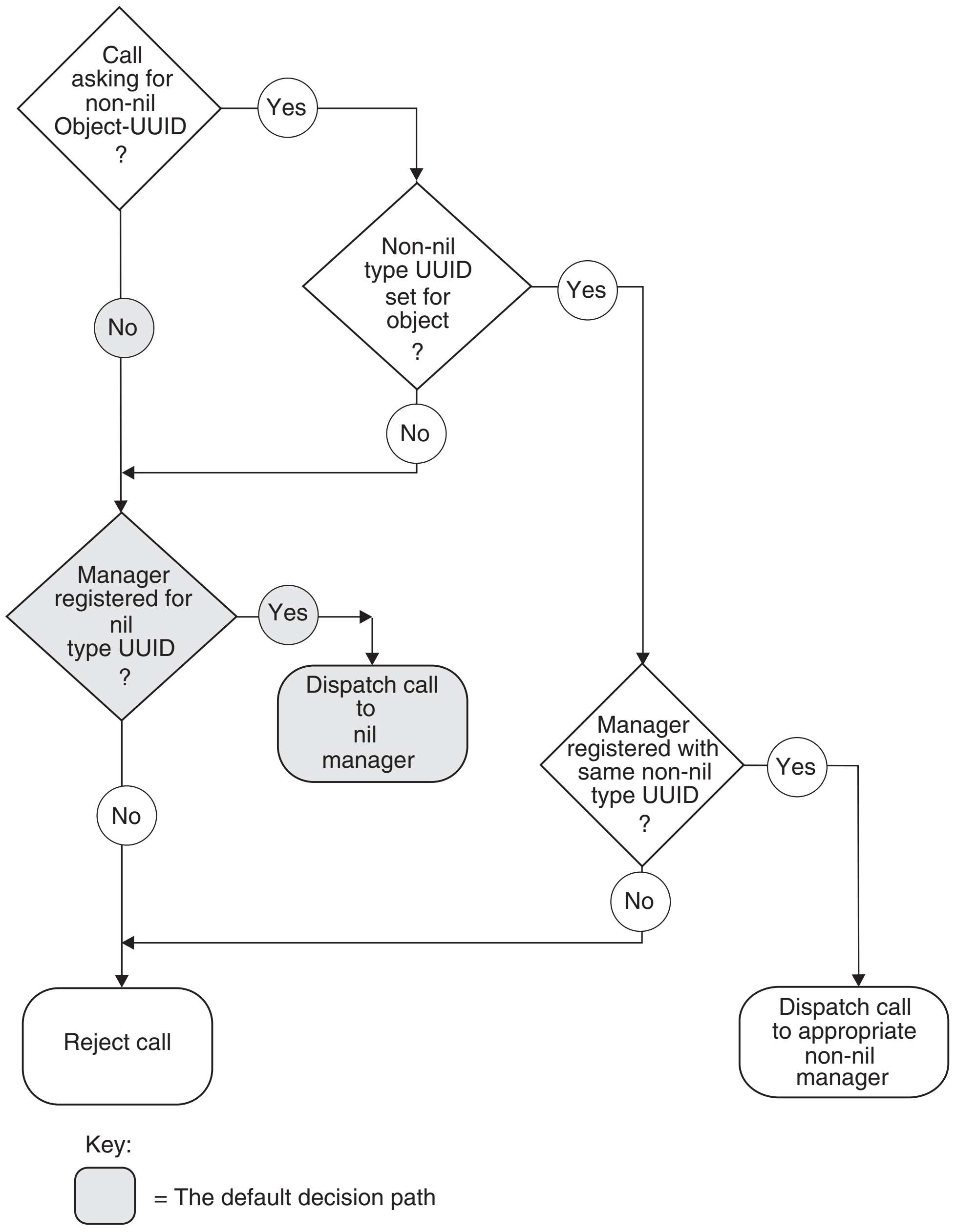

Call asking for non-nil Object-UUID ?
Yes
No
Non-nil type UUID set for object ?
Yes
No
Manager registered for nil type UUID ?
Yes
No
Dispatch call to nil manager
Manager registered with same non-nil type UUID ?
Yes
No
Reject call
Dispatch call to appropriate non-nil manager
Key:
= The default decision path

*Figure 47. Decisions for Selecting a Manager*

## Creating Portable Data Using the IDL Encoding Services

The IDL encoding services provide client and server RPC applications with a method for encoding data types in input parameters into byte stream format and decoding data types in output parameters from a byte stream without invoking the RPC runtime. Encoding and decoding functions are analogous to marshalling and unmarshalling, except that the data is stored locally, and is not transmitted over the network; the IDL encoding services separate the data marshalling and unmarshalling functions from interaction with the RPC runtime.

Client and server applications can use the IDL encoding services to "flatten" (or "serialize") a data structure, even binary data, and then store it, for example, by writing it to a file on disk. An RPC application on any DCE machine, regardless of its data type size and byte endianness, is then able to use the IDL encoding services to decode previously encoded data. Without the IDL encoding services, you cannot create a file of data on one machine and then successfully read that data on another machine that has different size data types and byte endianness.

The IDL encoding services can generate code that takes the input parameters to a procedure and places them in a standard form in one or more buffers that are delivered to user code. This process is called encoding. Encoded data can be written to a file. The IDL encoding services can also generate code that delivers, as the output parameters of a procedure, data that has been converted into the standard form by encoding. Delivery of data in this way is called decoding. Data to be decoded can be read from a file.

Applications use the ACF attributes **encode** and **decode** as operation attributes or as interface attributes to direct the IDL compiler to generate IDL encoding services stubs for operations rather than generating RPC stubs. See Chapter 12, “Attribute Configuration Language” on page 285 for usage information on **encode** and **decode**.

## Memory Management for IDL Encoding Services

IDL encoding services stubs handle memory management in the same way as RPC client stubs. When you call an operation to which the **encode** or **decode** or both attributes have been applied, the encoding services stub uses whatever client stub memory management scheme is currently in effect. “Memory Management” on page 173 gives further details on client stub memory management defaults and setting up memory management schemes.

You can control which memory management scheme the stubs will use by calling the **rpc_ss_swap_client_alloc_free()** and **rpc_ss_set_client_alloc_free()** routines. The first routine sets the memory management routines used by both the encoding and decoding stubs, and the second routine restores the previous memory management scheme after encoding and decoding are complete.

Note that the memory management scheme established, whether explicitly or by default, is on a per-thread basis.

## Buffering Styles

There are a number of different ways in which buffers containing encoded data can be passed between the application code and the IDL encoding services. These are referred to as different "buffering styles". The different buffering styles are:

| | |
|---|---|
| Incremental encoding | The incremental encoding style requires that you provide an *allocate* routine which creates an empty buffer into which IDL Encoding Services can place encoded data and a *write* routine which IDL encoding services will call when the buffer is full or all the parameters of the operation have been encoded. The IDL encoding services call the *allocate* and *write* routines repeatedly until the encoding of all of the parameters has been delivered to the user code. See the *z/OS DCE Application Development Reference* for a description of the required parameters for the *allocate* and *write* routines. |
| Fixed buffer encoding | The fixed buffer encoding style requires that the application supply a single buffer into which all the encoded data is to be placed. The buffer must have an address that is 8-byte aligned and must be a multiple of 8 bytes in size. It must also be large enough to hold an encoding of all the data, together with an encoding header for each operation whose parameters are being encoded; 56 bytes should be allowed for each encoding header. |

Dynamic buffer encoding

With the dynamic buffer encoding style, the IDL encoding services build a single buffer containing all the encoded data and deliver the buffer to application code. The buffer is allocated by whatever client memory management mechanism has been put in place by the application code. The default for this is **malloc**(). When the application code no longer needs the buffer, it should release the memory resource.

The dynamic buffer encoding style has performance implications. The IDL encoding services will usually allocate a number of intermediate buffers, then allocate the buffer to be delivered to the application code, copy data into it from the intermediate buffers, and release the intermediate buffers.

Incremental decoding

The incremental decoding buffering style requires that you provide a *read* routine which, when called, delivers to the IDL encoding services a buffer that contains the next part of the data to be decoded. The IDL encoding services will call the *read* routine repeatedly until all of the required data has been decoded. See the *z/OS DCE Application Development Reference* for a description of the required parameters for the *read* routine.

Buffer decoding

The buffer decoding style requires that you supply a single buffer containing all the encoded data. Where application performance is important, note that if the supplied buffer is not 8-byte aligned, the IDL encoding services allocate a temporary aligned buffer of comparable size and copy data from the user-supplied buffer into it before performing the requested decoding.

## IDL Encoding Services Handles

When an application's encoding or decoding operation is invoked, the handle passed to it must be an IDL encoding services handle (the **idl_es_handle_t** type). The IDL encoding services handle indicates whether encoding or decoding is required, and what style of buffering is to be used. The IDL encoding services provides a set of routines to enable the application code to obtain encoding and decoding handles to the IDL encoding services. The IDL encoding services handle-returning routine you call depends on the buffering style you have chosen:

- If you have selected the incremental encoding style, you call the **idl_es_encode_incremental()** routine, which returns an incremental encoding handle.
- If you have selected the fixed buffer encoding style, you call the **idl_es_encode_fixed_buffer()** routine, which returns a fixed buffer encoding handle.
- If you have selected dynamic buffer encoding, you call the **idl_es_encode_dyn_buffer()** routine, which returns a dynamic buffer encoding handle.
- If you have selected incremental decoding as your buffering style, you call the **idl_es_decode_incremental()** routine, which returns an incremental decoding handle.
- If you have selected the buffer decoding style, you call the **idl_es_decode_buffer()** routine, which returns a buffer decoding handle.

When the encoding or decoding for which an IDL encoding services handle was required is completed, the application code should release the handle resources by calling the **idl_es_handle_free()** routine. See the *z/OS DCE Application Development Reference* for a complete description of the IDL encoding service routines.

It is an error to call an operation for which **encode** or **decode** has been specified using an RPC binding handle, and it is an error to call an RPC operation using an IDL encoding services handle.

The following restrictions apply to the use of IDL encoding services handles:

- An operation can be called with an encoding handle only if the operation has been given the **encode** ACF attribute
- An operation can be called with a decoding handle only if the operation has been given the **decode** ACF attribute
- The **auto_handle** ACF attribute cannot be used with the IDL encoding services
- The **implicit_handle** ACF attribute cannot be used with the IDL encoding services
- Customized handles cannot be used with the IDL encoding services
- An **in** context handle does not contain the handle information needed by the IDL encoding services

## Programming Example

The following example uses the IDL encoding service features described in the preceding sections. The example verifies that the results of a number of decoding operations are the same as the parameters used to create the corresponding encodings.

The interface definition for this example is as follows:

```
[uuid(20aac780-5398-11c9-b996-08002b13d56d), version(0)]
interface es_array
{
    const long N = 5000;

    typedef struct
    {
        byte b;
        long l;
    } s_t;

    typedef struct
    {
        byte b;
        long a[7];
    } t_t;

    void in_array_op1([in] handle_t h, [in] long arr[N]);
    void out_array_op1([in] handle_t h, [out] long arr[N]);

    void array_op2([in] handle_t h, [in,out] s_t big[N]);

    void array_op3([in] handle_t h, [in,out] t_t big[N]);
}
```

The attribute configuration file for the example is as follows:

```
interface es_array
{
    [encode] in_array_op1();
    [decode] out_array_op1();
    [encode, decode] array_op2();
    [encode, decode] array_op3();
}
```

The test code for the example is as follows:

```
#include <dce/pthread_exc.h>
#include "rpcexc.h"
#include <stdio.h>
#include <stdlib.h>
#include <file.h>
```

```
#else
#include <sys/file.h>
#endif
#include "es_array.h"

/*
 *  User state for incremental encode/decode
 */
typedef struct es_state_t {
    idl_byte *malloced_addr;
    int file_handle;
} es_state_t;

static es_state_t es_state;

#define OUT_BUFF_SIZE 2048
static idl_byte out_buff[OUT_BUFF_SIZE];
static idl_byte *out_data_addr;
static idl_ulong_int out_data_size;

/*
 *  User allocate routine for incremental encode
 */
void es_allocate(state, buf, size)
idl_void_p_t state;
idl_byte **buf;
idl_ulong_int *size;
{
    idl_byte *malloced_addr;
    es_state_t *p_es_state = (es_state_t *)state;

    malloced_addr = (idl_byte *)malloc(*size);
    p_es_state->malloced_addr = malloced_addr;
    *buf = (idl_byte *)(((malloced_addr - (idl_byte *)0) + 7) & (~7));
    *size = (*size - (*buf - malloced_addr)) & (~7);
}

/*
 *  User write routine for incremental encode
 */
void es_write(state, buf, size)
idl_void_p_t state;
idl_byte *buf;
idl_ulong_int size;
{
    es_state_t *p_es_state = (es_state_t *)state;

    write(p_es_state->file_handle, buf, size);
    free(p_es_state->malloced_addr);
}

/*
 *  User read routine for incremental decode
 */
void es_read(state, buf, size)
idl_void_p_t state;
idl_byte **buf;
idl_ulong_int *size;
{
    es_state_t *p_es_state = (es_state_t *)state;

    read(p_es_state->file_handle, out_data_addr, out_data_size);
    *buf = out_data_addr;
    *size = out_data_size;
}

static ndr_long_int arr[N];
static ndr_long_int out_arr[N];
static s_t sarr[N];
static s_t ref_sarr[N];
static s_t out_sarr[N];
static t_t tarr[N];
static t_t ref_tarr[N];
static t_t out_tarr[N];
static ndr_long_int (*oarr)[M];

#define FIXED_BUFF_STORE (8*N+64)
static idl_byte fixed_buff_area[FIXED_BUFF_STORE];

/*
 *  Test Program
 */
main()
{
    idl_es_handle_t es_h;
    idl_byte *fixed_buff_start;
    idl_ulong_int fixed_buff_size, encoding_size;
    idl_byte *dyn_buff_start;
    error_status_t status;
    int i,j;


    for (i = 0; i < N; i++)
    {
        arr[i] = random()%10000;
        sarr[i].b = i & 0x7f;
        sarr[i].l = random()%10000;
        ref_sarr[i] = sarr[i];
        tarr[i].b = i & 0x7f;
        for (j = 0; j < 7; j++) tarr[i].a[j] = random()%10000;
        ref_tarr[i] = tarr[i];
    }

    /*
     *Incremental encode/decode
     */
    /* Encode data using one operation */
    es_state.file_handle = open("es_array_1.dat", O_CREAT|O_TRUNC|O_WRONLY, 0777);
    if (es_state.file_handle < 0)
    {
        printf("Can't open es_array_1.dat\n");
        exit(0);
    }
    idl_es_encode_incremental((idl_void_p_t)&es_state, es_allocate, es_write,
                            &es_h, &status);
    if (status != error_status_ok)
    {
        printf("Error %08x from idl_es_encode_incremental\n", status);
        exit(0);
    }
    in_array_op1(es_h, arr);
    close(es_state.file_handle);
    idl_es_handle_free(&es_h, &status);
    if (status != error_status_ok)
    {
        printf("Error %08x from idl_es_handle_free\n", status);
        exit(0);
    }

    /* Decode the data using another operation with the same signature */
    out_data_addr = (idl_byte *)(((out_buff - (idl_byte *)0) + 7) & (~7));
    out_data_size = (OUT_BUFF_SIZE - (out_data_addr - out_buff)) & (~7);
    es_state.file_handle = open("es_array_1.dat", O_RDONLY, 0);
    if (es_state.file_handle < 0)
    {
        printf("Can't open es_array_1.dat for reading\n");
        exit(0);
    }
    idl_es_decode_incremental((idl_void_p_t)&es_state, es_read,
                                &es_h, &status);
    if (status != error_status_ok)
    {
        printf("Error %08x from idl_es_decode_incremental\n", status);
        exit(0);
    }
    out_array_op1(es_h, out_arr);
    close(es_state.file_handle);
    idl_es_handle_free(&es_h, &status);
    if (status != error_status_ok)
    {
        printf("Error %08x from idl_es_handle_free\n", status);
        exit(0);
    }

    /* Check the input and output are the same */
    for (i = 0; i < N; i++)
    {
        if (out_arr[i] != arr[i])
        {
            printf("out_arr[%d] - found %d - expecting %d\n",
                    i, out_arr[i], arr[i]);
        }
    }

    /*
       * Fixed buffer encode/decode
       */
    fixed_buff_start = (idl_byte *)(((fixed_buff_area - (idl_byte *)0) + 7)
                                                              & (~7));
    fixed_buff_size = (FIXED_BUFF_STORE - (fixed_buff_start - fixed_buff_area))
                                                              & (~7);
    idl_es_encode_fixed_buffer(fixed_buff_start, fixed_buff_size,
                                &encoding_size, &es_h, &status);
    if (status != error_status_ok)
    {
        printf("Error %08x from idl_es_encode_fixed_buffer\n", status);
        exit(0);
    }
    array_op2(es_h, sarr);
    idl_es_handle_free(&es_h, &status);
    if (status != error_status_ok)
    {
        printf("Error %08x from idl_es_handle_free\n", status);
        exit(0);
    }
    idl_es_decode_buffer(fixed_buff_start, encoding_size, &es_h, &status);
    if (status != error_status_ok)
    {
        printf("Error %08x from idl_es_decode_buffer\n", status);
        exit(0);
    }
```

```
array_op2(es_h, out_sarr);
idl_es_handle_free(&es_h, &status);
if (status != error_status_ok)
{
    printf("Error %08x from idl_es_handle_free\n", status);
    exit(0);
}
for (i = 0; i < N; i++)
{
    if (out_sarr[i].b != ref_sarr[i].b)
    {
        printf("array_op2 - out_sarr[%d].b = %c\n", i, out_sarr[i].b);
    }
    if (out_sarr[i].l != ref_sarr[i].l)
    {
        printf("array_op2 - out_sarr[%d].l = %d\n", i, out_sarr[i].l);
    }
}

/*
 * Dynamic buffer encode - fixed buffer decode
 */
idl_es_encode_dyn_buffer(&dyn_buff_start, &encoding_size, &es_h, &status);
if (status != error_status_ok)
{
    printf("Error %08x from idl_es_encode_dyn_buffer\n", status);
    exit(0);
}
array_op3(es_h, tarr);
idl_es_handle_free(&es_h, &status);
if (status != error_status_ok)
{
    printf("Error %08x from idl_es_handle_free\n", status);
    exit(0);
}
idl_es_decode_buffer(dyn_buff_start, encoding_size, &es_h, &status);
if (status != error_status_ok)
{
    printf("Error %08x from idl_es_decode_buffer\n", status);
    exit(0);
}
array_op3(es_h, out_tarr);
rpc_ss_free (dyn_buff_start);
idl_es_handle_free(&es_h, &status);
if (status != error_status_ok)
{
    printf("Error %08x from idl_es_handle_free\n", status);
    exit(0);
}
for (i = 0; i < N; i++)
{
    if (out_tarr[i].b != ref_tarr[i].b)
    {
        printf("array_op3 - out_tarr[%d].b = %c\n", i, out_tarr[i].b);
    }
    for (j=0; j<7; j++)
    {
        if (out_tarr[i].a[j] != ref_tarr[i].a[j])
        {
            printf("array_op3 - out_tarr[%d].a[%d] = %d\n",
                    i, j, out_tarr[i].a[j]);
        }
    }
}
```

```
    printf("Test Complete\n");
}
```

## Performing Multiple Operations on a Single Handle

Multiple operations can be performed using one encoding handle before the handle is released. In this case, all the encoded data is part of the same buffer system.

A single decoding handle is used to obtain the contents of the encoded data. Decoding operations must be called in the same order the encoding operations were called to create the encoded data.

The definition of the user client memory management functions, and any memory allocated by IDL encoding services using the client memory allocator, must not be modified between operations for which the same encoding handle is used.

## Determining the Identity of an Encoding

Applications can use the **idl_es_inq_encoding_id()** routine to determine the identity of an encoding operation, for example, before calling their decoding operations.

# Chapter 11. Interface Definition Language

This chapter describes how to construct an Interface Definition Language (IDL) file. First, it describes the IDL syntax notation conventions and lexical elements. It then describes the interface definition structure and the individual language elements supported by the IDL compiler.

**Note:** In z/OS DCE, IDL provides services to facilitate the operations of applications using RPC. The following examples, which relate to topics discussed in this chapter, can be found in the **/usr/lpp/dce/examples/idl** directory:

- es_array
- n_e_union
- pipes
- refdels
- strint
- ue1
- unique1

## The Interface Definition Language File

The Interface Definition Language (IDL) file defines all aspects of an interface that affect data passed over the network between a caller (client) and a callee (server). The interface definition file usually has a file extension of 'IDL', or a qualifier 'IDL' if it is a data set. For a caller and callee to interoperate, they both need to incorporate the same interface definition.

IDL files used with z/OS DCE must be coded in code page IBM-1047. The output files from the z/OS DCE IDL compiler (header files, client and server stub files) will also be in code page IBM-1047.

## Syntax Notation Conventions

In addition to the documentation conventions described in the Preface of this book, the IDL syntax uses the special notation described in the following subsections.

### Typography

IDL documentation uses the following type style:

| | |
|---|---|
| **bold** | Bold type style indicates a literal item. Keywords and literal punctuation are represented in bold type style. Identifiers used in a particular example are represented in bold type style when mentioned in the text. |
| *italic* | Italic type style indicates a symbolic item for which you need to substitute a particular value. In IDL syntax descriptions, all identifiers that are not keywords are represented in italic type style. |
| `monospace` | A monospace type style is used for source code examples (in IDL or in C) that are displayed separately from regular text. |

## Special Symbols

IDL documentation uses the following symbolic notations:

| | |
|---|---|
| *[item]* | *Italic* brackets surrounding an item indicate that the item is optional. |
| [*item*] | Brackets shown in regular type style are a required part of the syntax. |
| *item* . . . | Ellipsis dots following an item indicate that the item may occur one or more times. |
| *item* **, ...** | If an item is followed by a literal punctuation character and then by ellipsis points, the item may occur either once without the punctuation character, or more than once, with the punctuation character separating each instance. |
| . . . | If ellipsis points are shown on a line by themselves, the item or set of items in the preceding line may occur any number of additional times. |
| *item* \| *item* | If several items are shown separated by vertical bars, exactly one of those items must occur. |

## IDL Lexical Elements

The following subsections describe these IDL lexical elements:

- Identifiers
- Keywords
- Punctuation characters
- White space
- Case sensitivity

### Identifiers

The character set for IDL identifiers comprises the alphabetic characters A to Z and a to z, the digits 0 to 9, and the _ (underscore) character. An identifier must start with an alphabetic character.

No IDL identifier can exceed 31 characters. In some cases, an identifier has a shorter maximum length because the IDL compiler uses the identifier as a base from which to construct other identifiers; each case is identified as it occurs.

### Keywords

IDL reserves some identifiers as keywords. In the text of this chapter, keywords are represented in **bold** type style, and identifiers that you choose are represented in *italic* type style.

### Punctuation Characters

IDL uses the following graphic characters:

```
"  '  (  )  *  ,  .  /  :  ;  | =  [  \  ]  {  }  ^  ˜
```

In addition, the following trigraphs are supported by z/OS DCE as an alternative to entering special characters:

| **Trigraph** | **Special character** |
|---|---|
| `??(` | Left square bracket ([) |
| `??)` | Right square bracket (]) |

| | |
|---|---|
| `??'` | Caret (^) |
| `??!` | Vertical bar (\|) |
| `??-` | Tilde (˜) |
| `??<` | maybe be substituted for a left brace if the **{** (left brace) national replacement set character is not available on the keyboard. |
| `??>` | maybe be substituted for a right brace if the **{** (right brace) national replacement set character is not available on the keyboard. |

Use of these trigraph sequences adds the following punctuation characters to the set in the preceding list:

```
< > ? ! -
```

## White Space

White space is used to delimit other constructs. IDL defines the following white space constructs:

- A space
- A carriage return
- A horizontal tab
- A form feed at the beginning of a line
- A comment
- A sequence of one or more of the preceding white space constructs

A keyword, identifier, or number not preceded by a punctuation character must be preceded by white space. A keyword, identifier, or number not followed by a punctuation character must be followed by white space. Unless noted otherwise, any punctuation character may be preceded and possibly followed by white space.

When enclosed in **" "** (double quotation marks) or **' '** (single quotation marks), white space constructs are treated literally. Otherwise, they serve only to separate other lexical elements and are ignored.

The character sequence **/*** (slash and asterisk) begins a comment, and the character sequence ***/** (asterisk and slash) ends a comment. For example:

```
/* all natural */
import "potato.idl";  /* no preservatives */
```

Comments cannot be nested.

## Case Sensitivity

The IDL language is case sensitive. The IDL compiler does not force the case of identifiers in the generated code. You only have to be aware of the implications involved in calling generated stubs from languages other than C. File names are not case sensitive.

## IDL versus C

IDL resembles a subset of ANSI C. The major difference between IDL and C language is that there are no executable statements in IDL.

## Declarations

An interface definition specifies how operations are called, not how they are carried out. IDL is therefore a purely declarative language.

## Data Types

To support applications written in languages other than C, IDL defines some data types that do not exist in C and extends some data types that do exist in C. For example, IDL defines a Boolean data type.

Some C data types are supported by IDL only with modifications or restrictions. For example, unions must be discriminated, and all arrays must be accompanied by bounds information.

## Attributes

The stub modules that are generated from an interface definition require more information about the interface than can be expressed in C. For example, stubs must know whether an operation parameter is an input or an output.

The additional information required to define a network interface is specified using IDL attributes. IDL attributes can apply to types, to structure members, to operations, to operation parameters, or to the interface as a whole. Some attributes are allowed in only one of the preceding contexts; others are allowed in more than one context. An attribute is always represented in **[ ]** (brackets) before the item to which it applies. For example, in an operation declaration, inputs of the operation are preceded by the **in** attribute and outputs are preceded by the **out** attribute:

```
void arith_add (
     [in] long a,
     [in] long b,
     [out] long *c,
     );
```

## Interface Definition Structure

An interface definition has the following structure:

> **[***interface_attribute***, ...] interface** *interface_name*
> **{**
> *declarations*
> **}**

The portion of an interface definition that precedes the **{** (left brace) is the interface header. The remainder of the definition is the interface body. Interface header syntax and interface body syntax are described separately in the following two subsections.

## Interface Definition Header

The interface header comprises a list of interface attributes enclosed in **[ ]** (brackets), the keyword **interface**, and the interface name:

> **[***interface_attribute***, ...] interface** *interface_name*

Interface names, together with major and minor version numbers, are used by the IDL compiler to construct identifiers for interface specifiers, entry point vectors, and entry point vector types. If the major and minor version numbers are single digits, the interface name can be up to 17 characters long.

## Interface Definition Body

The *declarations* in an interface definition body are one or more of the following:

*import_declaration*
*constant_declaration*
*type_declaration*
*operation_declaration*

A **;** (semicolon) ends each declaration, and **{ }** (braces) enclose the entire body.

Import declarations must precede other declarations in the interface body. Import declarations specify the names of other IDL interfaces that define types and constants used by the importing interface.

Constant, type, and operation declarations specify the constants, types, and operations that the interface exports. These declarations can be coded in any order, provided any constant or type is defined before it is used.

## Overview of IDL Attributes

Table 9 lists the attributes allowed in interface definition files and specifies the declarations in which they can occur.

*Table 9. IDL Attributes*

| Attribute | Where Used |
|---|---|
| **uuid**<br>**version**<br>**endpoint**<br>**exceptions**<br>**pointer_default**<br>**local** | Interface definition headers |
| **broadcast**<br>**maybe**<br>**idempotent**<br>**reflect_deletions** | Operations |
| **in**<br>**out** | Parameters |
| **ignore** | Structures |
| **max_is**<br>**min_is**<br>**size_is**<br>**first_is**<br>**last_is**<br>**length_is** | Arrays |
| **string** | Arrays |
| **ptr**<br>**ref** | Pointers |
| **handle** | Customized handles |
| **context_handle** | Context handles |
| **transmit_as** | Type declarations |

## Interface Definition Header Attributes

This following subsections describe in detail the usage and semantics of the IDL attributes that can be used in interface definition headers. The attributes provided for interface definition headers are as follows:

- **uuid**
- **version**
- **endpoint**
- **exceptions**
- **pointer_default**
- **local**

## The uuid Attribute

The **uuid** attribute specifies the Universal Unique Identifier (UUID) that is assigned to an interface. The **uuid** attribute takes the form:

**uuid (***uuid_string***)**

A *uuid_string* is the string representation of a UUID. This string is typically generated as part of a skeletal interface definition by the utility **uuidgen**. A *uuid_string* contains one group of 8 hexadecimal digits, three groups of 4 hexadecimal digits, and one group of 12 hexadecimal digits, with hyphens separating the groups, as in the following example:

```
01234567-89ab-cdef-0123-456789abcdef
```

A new UUID should be generated for any new interface. If several versions of one interface exist, all versions should have the same interface UUID but different version numbers. A client and a server cannot communicate unless the interface imported by the client and the interface exported by the server have the same UUID. The client and server stubs in an application must be generated from the same interface definition or from interface definitions with identical **uuid** attributes.

Any remote interface must have the **uuid** attribute. An interface containing operation definitions must have either the **uuid** attribute or the **local** attribute, but cannot have both.

The **uuid** attribute can appear at most once in an interface.

The following example illustrates use of the **uuid** attribute:

```
uuid(4ca7b4dc-d000-0d00-0218-cb0123ed9876)
```

## The version Attribute

The **version** attribute specifies a particular version of a remote interface. The **version** attribute takes the form:

**version (***major [.minor ]***)**

A version number can be either a pair of integers (the major and minor version numbers) or a single integer (the major version number). If both major and minor version numbers are supplied, the integers should be separated by a period without white space. If no minor version number is supplied, 0 (zero) is assumed.

The following examples illustrate use of the **version** attribute:

```
version (1.1)   /* major and minor version numbers */
version (3)     /* major version number only */
```

The **version** attribute can be omitted altogether, in which case the interface is assigned 0.0 as the default version number.

A client and a server can communicate only if the following requirements are met:

- The interface imported by the client and the interface exported by the server have the same major version number.
- The interface imported by the client has a minor version number less than or equal to that of the interface exported by the server.

You must increase the major version number when you make any incompatible change to an interface definition. (See the definition of compatible changes that follows.) You cannot decrease the major version number.

The following are considered compatible changes to an interface definition:

- Adding operations to the interface, if and only if the new operations are declared after all existing operation declarations in the interface definition.
- Adding type and constant declarations, if the new types and constants are used only by operations added at the same time or later. Existing operation declarations *cannot* have their signatures changed.

You should increase the minor version number only when you make a compatible change to an interface definition. You must not decrease the minor version number unless you simultaneously increase the major version number.

The *major* and *minor* integers in the **version** attribute can range from 0 to 65,535 inclusive. However, these typically are small integers and are increased in increments of one.

The following are considered incompatible changes to an interface definition:

- Changing the signature of an existing operation
- Changing the order of existing operations
- Adding a new operation other than at the end.

The **version** attribute can appear at most once in an interface.

## The endpoint Attribute

The **endpoint** attribute specifies the well-known endpoint or endpoints on which servers that export the interface will listen. The **endpoint** attribute takes the form:

**endpoint (***endpoint_spec***, ...)**

Each *endpoint_spec* is a string in the following form:

**"** *family* **: [***endpoint***] "**

The *family* identifies a protocol family. The following are example values for *family*:

| | |
|---|---|
| ncacn_ip_tcp | NCA Connection over Internet Protocol: Transmission Control Protocol (TCP/IP) |
| ncadg_ip_udp | NCA Datagram over Internet Protocol: User Datagram Protocol (UDP/IP) |

The *endpoint* identifies a well-known endpoint for the specified *family*. The values accepted for *endpoint* depend on the *family* but typically are integers within a limited range. IDL does not define valid *endpoint* values.

Well-known endpoint values are typically assigned by the central authority that *owns* a protocol. For example, the Internet Assigned Numbers Authority assigns well-known endpoint values for the IP protocol family.

At compile time, the IDL compiler checks each *endpoint_spec* only for gross syntax. At run time, stubs pass the *family* and *endpoint* strings to the RPC runtime, which validates and interprets them.

Most applications should not use well-known endpoints and should instead use dynamically assigned opaque endpoints. Most interfaces designed for use by applications should therefore not have the **endpoint** attribute.

The **endpoint** attribute can appear at most once in an interface. The same family cannot be used more than once within the endpoint attribute.

The following example illustrates use of the **endpoint** attribute:

```
endpoint ("ncadg_ip_udp:[6677]","ncacn_ip_tcp:[6677]")
```

## The exceptions Attribute

The **exceptions** attribute specifies a set of user-defined exceptions that can be generated by the server implementation of the interface. The **exceptions** attribute takes the form:

**exceptions (***exception_name* [**,***exception_name*] **...)**

The following is a sample declaration of an exceptions attribute:

```
[uuid(06255501-08af-11cb-8c4f-08002b13d56d),
version (1.1),
  exceptions (
      exc_e_exquota,
      binop_e_aborted,
      binop_e_too_busy,
      binop_e_shutdown)
] interface binop
  {
      long binop_add(
           [in] long a,
           [in] long b
           );
   }
```

See Chapter 10, "Topics in RPC Application Development" on page 173 for more information on using exceptions.

The **exceptions** attribute can appear at most once in an interface.

## The pointer_default Attribute

IDL supports two kinds of pointer semantics. The **pointer_default** attribute specifies the default semantics for pointers that are declared in the interface definition. The **pointer_default** attribute takes the form:

**pointer_default (***pointer_attribute***)**

Possible values for *pointer_attribute* are **ref**, **ptr** and **unique**.

The default semantics established by the **pointer_default** attribute apply to the following usages of pointers:

- A pointer that occurs in the declaration of a member of a structure or a union.
- A pointer that does not occur at the top level of an operation parameter declared with more than one pointer operator. A top-level pointer is one that is not the target of another pointer, and is not a field of a data structure, which is the target of a pointer. See "Pointer Attributes in Parameters" on page 257 for more information on top-level pointers.

**Note:** The **pointer_default** attribute does not apply to a pointer that is the return value of an operation, because this is always a full pointer.

The default semantics can be overridden by pointer attributes in the declaration of a particular pointer. If an interface definition does not specify **pointer_default** and contains a declaration that requires default pointer semantics, the IDL compiler will issue an error.

The **pointer_default** attribute can appear at most once in an interface.

For additional information on pointer semantics, refer to "Pointer Attributes" on page 254.

## The local Attribute

The **local** attribute indicates that an interface definition does not declare any remote operations and that the IDL compiler should therefore generate only header files, not stub files. The **local** attribute takes the form:

**local**

An interface containing operation definitions must have either the **local** attribute or the **uuid** attribute. No interface can have both.

If the **local** attribute is coded more than once in an interface, the IDL compiler issues a warning.

## Rules for Using Interface Definition Header Attributes

An interface cannot have both the **local** attribute and the **uuid** attribute. In an interface definition that contains any operation declarations, either **local** or **uuid** must be specified. In an interface definition that contains no operation declarations, both **local** and **uuid** can be omitted.

The **endpoint**, **pointer_default**, **exceptions**, **uuid**, and **version** attributes cannot be coded more than once. If the **local** attribute is coded more than once, the IDL compiler issues a warning.

## Examples of Interface Definition Header Attributes

The following example uses the **uuid** and **version** attributes:

```
[uuid(df961f80-2d24-11c9-be74-08002b0ecef1), version(1.1)]
interface my_interface_name
```

The following example uses the **uuid**, **endpoint**, and **version** attributes:

```
[uuid(0bb1a080-2d25-11c9-8d6e-08002b0ecef1),
endpoint("ncadg_ip_udp[6677]", "ncacn_ip_tcp:[6677]"),
version(3.2)]
interface my_interface_name
```

## Import Declarations

The IDL *import_declaration* specifies interface definition files that declare types and constants used by the importing interface. It takes the following form:

**import** *file*,... **;**

The *file* is either an HFS file name or PDS member name of the interface definition you are importing, enclosed in **" "** (double quotation marks). For HFS files, *file* can include a path name. This path name can be relative; the **-I** option of the IDL compiler allows you to specify a parent directory from which to resolve import path names. For PDS members, the **-I** option of the IDL compiler allows you to specify a DDname from which to resolve import files.

The effect of an import declaration is as if all constant, type, and import declarations from the imported file occurred in the importing file at the point where the import declaration occurs. Operation declarations are not imported.

For example, suppose that the interface definition **aioli.idl** contains a declaration to import the definitions for the **garlic** and **oil** interfaces:

```
import "garlic.idl", "oil.idl";
```

The IDL compiler will generate a C header file named **aioli.h** that contains the following **#include** directives:

```
#include "garlic.h"
#include "oil.h"
```

The stub files that the compiler generates will not contain code for any **garlic** and **oil** operations.

Importing an interface many times has the same effect as importing it once.

## Constant Declarations

The IDL *constant_declaration* can take the following forms:

**const** *integer_type_spec identifier* **=** *integer* | *value* | *integer_const_expression***;**
**const boolean** *identifier* **= TRUE** | **FALSE** | *value***;**
**const char** *identifier* **=** *character* | *value***;**
**const char*** *identifier* **=** *string* | *value***;**
**const void*** *identifier* **= NULL** | *value***;**

The *integer_type_spec* is the data type of the integer constant you are declaring. The *identifier* is the name of the constant. The *integer*, *integer_const_expression*, *character*, *string*, or *value* specifies the value to be assigned to the constant. A *value* can be any previously defined constant.

IDL provides only integer, Boolean, character, string, and null pointer constants.

Following are examples of constant declarations:

```
const short TEN = 10;
const boolean FAUX = FALSE;
const char* DSCH = "Dmitri Shostakovich";
```

## Integer Constants

An *integer_type_spec* is a *type_specifier* for an integer, except that the *int_size* for an integer constant cannot be **hyper**.

An *integer* is the decimal representation of an integer. IDL also supports the C notation for hexadecimal, octal, and long integer constants.

You can specify any previously defined integer constant as the *value* of an integer constant.

You can specify any arithmetic expression as the *integer_const_expression* that evaluates to an integer constant.

## Boolean Constants

A Boolean constant can take one of two values: TRUE or FALSE.

You can specify any previously defined Boolean constant as the *value* of a Boolean constant.

## Character Constants

A *character* is a character enclosed in **' '** (single quotation marks). White space characters are interpreted literally. The backslash character '\' introduces an escape sequence, as defined in the ANSI C standard. The single quotation mark character '**'**' can be coded as the *character* only if it is escaped by a backslash.

You can specify any previously defined character constant as the *value* of a character constant.

## String Constants

A *string* is a sequence of characters enclosed in " " (double quotation marks). White space characters are interpreted literally. The **** (backslash) character introduces an escape sequence, as defined in the ANSI C standard. The **"** (double quotation mark) character can be coded in a *string* only if it is escaped by a backslash.

You can specify any previously defined string constant as the *value* of a string constant.

## NULL Constants

A **void*** constant can take only one literal value: **NULL**.

You can specify any previously defined **void*** constant as the *value* of a **void*** constant.

## Type Declarations

The IDL *type_declaration* enables you to associate a name with a data type and to specify attributes of the data type. It takes the following form:

> **typedef** *[***[***type_attribute***, ...]***] type_specifier type_declarator***, ...;**

A *type_attribute* specifies characteristics of the type being declared.

The *type_specifier* can specify a base type, a constructed type, a predefined type, or a named type.

A function pointer can be specified if the *local* attribute has been specified.

Each *type_declarator* is a name for the type being defined. Note, though, that a *type_declarator* can also be preceded by an ***** (asterisk), followed by **[ ]** (brackets), and can include **( )** (parentheses) to indicate the precedence of its components.

## Type Attributes

A *type_attribute* can be any of the following:

| | |
|---|---|
| **handle** | The type being declared is a user-defined, customized-handle type. |
| **context_handle** | The type being declared is a context-handle type. |
| **transmit_as** | The type being declared is a *presented type*. When it is passed in RPCs, it is converted to a specified *transmitted type*. |
| **ref** | The type being declared is a reference pointer. |
| **ptr** | The type being declared is a full pointer. |
| **unique** | The type being declared is a unique pointer. |
| **string** | The array type being declared is a string type. |

## Base Type Specifiers

IDL base types include integers, floating-point numbers, characters, a boolean type, a byte type, a void type, and a primitive handle type.

The IDL base data type specifiers are listed in Table 10. Where applicable, the table shows the size of the corresponding transmittable type and the type macro emitted by the IDL compiler for resulting declarations.

*Table 10 (Page 1 of 2). Base Data Type Specifiers*

| | **Specifier** | | **Size** | **Type Macro** |
|---|---|---|---|---|
| **(sign)** | **(size)** | **(type)** | | **Emitted by idl** |
| | small | int | 8 bits | idl_small_int |
| | short | int | 16 bits | idl_short_int |

*Table 10 (Page 2 of 2). Base Data Type Specifiers*

| (sign) | Specifier (size) | (type) | Size | Type Macro Emitted by idl |
|---|---|---|---|---|
| | long | int | 32 bits | idl_long_int |
| | hyper | int | 64 bits | idl_hyper_int |
| unsigned | small | int | 8 bits | idl_usmall_int |
| unsigned | short | int | 16 bits | idl_ushort_int |
| unsigned | long | int | 32 bits | idl_ulong_int |
| unsigned | hyper | int | 64 bits | idl_uhyper_int |
| | | float | 32 bits | idl_short_float |
| | | double | 64 bits | idl_long_float |
| unsigned | | char | 8 bits | idl_char |
| | | char | 8 bits | idl_norm_char |
| | | boolean | 8 bits | idl_boolean |
| | | byte | 8 bits | idl_byte |
| | | void* | - | idl_void_p_t |
| | | handle_t | - | - |

The base types are described individually later in this chapter.

Note that you can use the **idl_** macros in the code you write for an application to ensure that your type declarations are consistent with those in the stubs, even when the application is transferred to another platform. The **idl_** macros are especially useful when passing constant values to RPC calls. For maximum portability, all constants passed to RPC calls declared in your network interfaces should be cast to the appropriate type, because the size of integer constants (like the size of the **int** data type) is ambiguous in the C language.

The **idl_** macros are defined in **<dce/idlbase.h>**, which is included by the header file that the IDL compiler generates.

**Note:** To conform to the type checking of the C/C++ compilers, a type macro **idl_norm_char** defines the **char** type declaration. The **idl_char** type macro defines only the **unsigned char** type declaration. All existing applications that used **idl_char** to define an input or output variable within the server or client application and used **char** to define the same variable in the IDL file may experience C compiler errors. Choose one of the following options to resolve these errors:

- Change **idl_char** to **idl_norm_char** within the server or client application.
- Change **char** to **unsigned char** within the IDL file and recompile the IDL file.
- Recompile the IDL file using the new **char_is_unsigned_char** IDL compiler option. This causes both **char** and **unsigned char** to be defined as **idl_char**. Refer to the **idl** command section of the *z/OS DCE Command Reference* for further explanation.

For all new applications, ensure that character variable types are consistent between the application and the IDL file.

| IDL | Application |
|---|---|
| char | char or idl_norm_char |
| unsigned char | unsigned char or idl_char |

## Constructed Type Specifiers

IDL constructed types include structures, unions, enumerations, pipes, arrays, and pointers. (In IDL, as in C, arrays and pointers are specified using declarator constructs rather than type specifiers.) Following are the keywords used to declare constructed type specifiers:

**struct**
**union**
**enum**
**pipe**

Constructed types are described in detail later in this chapter.

## Predefined Type Specifiers

While IDL does not have any predefined types, the DCE RPC IDL implicitly imports **nbase.idl**, which does predefine some types. Specifically, **nbase.idl** predefines an error status type, several international character data types, and many other types. The keywords used to declare these *predefined* type specifiers are:

**error_status_t**
**ISO_LATIN_1**
**ISO_MULTI_LINGUAL**
**ISO_UCS**

The error status type and international characters are described in detail later in this chapter.

## Type Declarator

An IDL *type_declarator* can be either a simple declarator or a complex declarator. A **simple declarator** is just an identifier. A **complex declarator** is an identifier that specifies an array, a function pointer, or a pointer.

## Operation Declarations

The IDL *operation_declaration* can take the following forms:

*[***[***operation_attribute***, ...]***] type_specifier operation_identifier*
**(***parameter_declaration***, ...);**

*[***[***operation_attribute***, ...]***] type_specifier operation_identifier*
**(***[***void***]***);**

Use the first form for an operation that has one or more parameters; use the second form for an operation that has no parameters.

An *operation_attribute* can take the following forms:

| | |
|---|---|
| **idempotent** | The operation is idempotent. |
| **broadcast** | The operation is always to be broadcast. |
| **maybe** | The caller of the operation does not require and will not receive any response. |

| | |
|---|---|
| **reflect_deletions** | If **rpc_ss_free()** is applied by application code on the server side to memory used for the referent of a full pointer that is part of an **[in]** parameter, the storage occupied by that referent on the client side is released. |
| **ptr** | The operation returns a full pointer. This attribute must be supplied if the operation returns a pointer result and reference pointers or unique pointers are the default for the interface. |
| **context_handle** | The operation returns a context handle. |
| **string** | The operation returns a string. |

**Note:** If any of the above operation attributes is coded more than once, the IDL compiler issues a warning.

The *type_specifier* in an operation declaration specifies the data type that the operation returns, if any. This type must be either a scalar type or a previously defined type. If the operation does not return a result, its *type_specifier* must be **void**.

For information on the semantics of pointers as operation return values, refer to “Pointers” on page 254.

The *operation_identifier* in an operation declaration is an identifier that names the operation.

Each *parameter_declaration* in an operation declaration declares a parameter of the operation. A *parameter_declaration* takes the following form:

**[** *parameter_attribute* **, . . . ]** *type_specifier parameter_declarator*

Parameter declarations and the parameter attributes are described separately in the following sections.

## Operation Attributes

Operation attributes determine the semantics to be applied by the RPC client and server protocol when an operation is called.

## Operation Attributes: idempotent, broadcast, and maybe

The **idempotent** attribute specifies that an operation is idempotent; that is, it can be run safely more than once.

The **broadcast** attribute specifies that an operation is to be broadcast to all hosts on the local network each time the operation is called. The client receives output arguments from the first reply to return successfully, and all subsequent replies are discarded.

An operation with the **broadcast** attribute is implicitly idempotent. Note that the broadcast capabilities of RPC runtime have a number of distinct limitations:

- Not all systems and networks support broadcasting. In particular, broadcasting is not supported by the RPC connection-oriented protocol.
- Broadcasts are limited to hosts on the local network.
- Broadcasts make inefficient use of network bandwidth and processor cycles.
- The RPC runtime library does not support **at-most-once** semantics for broadcast operations; it applies **idempotent** semantics to all such operations.
- The input arguments for broadcast calls are limited to 944 bytes.

The **maybe** attribute specifies that the caller of an operation does not expect any response. An operation with the **maybe** attribute cannot have any output parameters and cannot return anything. Delivery of the call is not ensured.

An operation with the **maybe** attribute is implicitly idempotent.

**Note:** Authenticated RPC for **maybe** calls is not supported on z/OS DCE.

Note that if you do not use any of the above operation attributes, the default is at-most-once semantics.

## Operation Attributes: Memory Management

Use the **reflect_deletions** attribute to mirror the release of memory from server pointer targets to client pointer targets. When you use the reflect_deletions attribute, memory occupied by pointer targets on the client will be released when the corresponding pointer targets on the server are released. This is only true for pointer targets that are components of **[in]** parameters of the operation. By default, the mechanism used by RPC to release the pointer targets is the C language free() function unless the client code is executing as part of RPC server application code, in which case the **rpc_ss_free()** function is used. You can override the default by calling **rpc_ss_set_client_alloc_free()** or **rpc_ss_swap_client_alloc_free()** before the call to the remote operation.

## Parameter Declarations

A *parameter_declaration* is used in an operation declaration to declare a parameter of the operation. A *parameter_declaration* takes the form:

**[***parameter_attribute***, ...]** *type_specifier parameter_declarator*

If an interface neither uses implicit handles nor an interface-based binding, the first parameter must be an explicit handle that gives the object UUID and location. The handle parameter can be of a primitive handle type, **handle_t**, or a nonprimitive user-defined handle type.

A *parameter_attribute* can be any of the following:

| | |
|---|---|
| *array_attribute* | One of several attributes that specifies the characteristics of arrays. |
| **in** | The parameter is an input attribute. |
| **out** | The parameter is an output attribute. |
| **ref** | The parameter is a reference pointer; it cannot be **NULL** and cannot be an alias. |
| **unique** | The parameter is a unique pointer; it can be **NULL** but cannot be an alias. |
| **ptr** | The parameter is a full pointer; it can be **NULL** and can be an alias. |
| **string** | The parameter is a string. |
| **context_handle** | The parameter is a context handle. |

The directional attributes **in** and **out** specify the directions in which a parameter is to be passed. The **in** attribute specifies that the parameter is passed from the caller to the callee. The **out** attribute specifies that the parameter is passed from the callee to the caller.

An output parameter must be passed by reference and therefore must be declared with an explicit *****. (Note that an array is implicitly passed by reference and so an output array does not require an explicit *****.) At least one directional attribute must be specified for each parameter of an operation.

An explicit handle parameter must have at least the **in** attribute.

The **ref**, **ptr** and **unique** attributes are described in “Pointers” on page 254. The **string** attribute is described in “Arrays” on page 247 and “Strings” on page 253. The **context_handle** attribute is described in “Context Handles” on page 268.

The *type_specifier* in a parameter declaration specifies the data type of the parameter.

The *declarator* in a parameter declaration can be any simple or complex declarator.

A parameter with the **out** attribute must be either an array or an explicitly declared pointer. An explicitly declared pointer is declared by a *pointer_declarator*, rather than by a *simple_declarator* with a named pointer type as its *type_specifier*.

For information on the semantics of pointers as operation parameters, refer to “Pointers” on page 254.

## Basic Data Types

This section describes the basic data types provided by IDL and the treatment of international characters within IDL. The basic data types are as follows:

- Integer types
- Floating-point types
- The **char** type
- The **boolean** type
- The **byte** type
- The **void** type
- The **handle_t** type
- The **error_status_t** type.

The following subsections describe the constructed data types that are built on the basic data types.

## Integer Types

IDL provides four sizes of signed and unsigned integer data types, specified as follows:

*int_size [***int***]*
**unsigned** *int_size [***int***]*
*int_size* **unsigned** *[***int***]*

The *int_size* can take the following values:

**hyper**
**long**
**short**
**small**

The **hyper** types are represented in 64 bits. A **long** is 32 bits. A **short** is 16 bits. A **small** is 8 bits.

The keyword **int** is optional and has no effect. The keyword **unsigned** denotes an unsigned integer type; it can occur either before or after the size keyword.

The **hyper** data type is a structure defined by DCE. It consists of 2 fields, a low field that represents the lower 32 bits and a high field that represents the higher 32 bits of a 64 bit hyper data type.

You can refer to hyper data as follows:

```
hyper h_var;

   h_var.low
   h_var.high
```

For a pointer, you can refer to it as:

```
hyper * h_var;

   h_var->low
   h_var->high
```

Consider a very large decimal number, 592,705,486,847. In hexadecimal representation, this number is X'00000089FFFFFFFF'. You can initialize and assign the hyper variable **h_var** to this number as follows:

```
hyper h_var;

       h_var.low  = 0xFFFFFFFF;
       h_var.high = 0x00000089;
```

When you manipulate data declared as hyper (for example, to add or multiply hyper data) you need to handle the overflow of the low field into the high field in your operation.

## Floating-Point Types

IDL provides two sizes of floating-point data types, specified as follows:

**float**
**double**

A **float** is represented in 32 bits. A **double** is represented in 64 bits.

## The char Type

The IDL character type is specified as follows:

*[***unsigned***]* **char**

A **char** is unsigned and is represented in 8 bits.

The keyword **unsigned** is optional.

**Important Note:** If **unsigned char** is used within an IDL definition, it must also be used within the client, server and manager code.

## The boolean Type

The IDL **boolean** type is specified as follows:

**boolean**

A **boolean** is represented in 8 bits. A **boolean** is a logical quantity that assumes one of two values: TRUE or FALSE. Zero is FALSE, and any nonzero value is TRUE.

## The byte Type

The IDL **byte** type is specified as follows:

**byte**

A **byte** is represented in 8 bits. The data representation format of **byte** data is guaranteed not to change when the data is transmitted by the RPC mechanism.

The IDL integer, character, and floating-point types (and hence any types constructed from these) are all subject to format conversion when they are transmitted between hosts that use different data representation formats. You can protect data of any type from format conversion by transmitting that type as an array of **byte**.

## The void Type

The IDL **void** type is specified as follows:

**void**

The **void** type may be used to do the following:

- Specify the type of an operation that does not return a value
- Specify the type of a context handle parameter, which must be **void***
- Specify the type of a **NULL** pointer constant, which must be **void***

## The handle_t Type

The IDL primitive handle type is specified as follows:

**handle_t**

A **handle_t** is a primitive handle type opaque to application programs but meaningful to the RPC runtime library. "Customized Handles" on page 267 discusses primitive and nonprimitive handle types.

## The error_status_t Type

IDL provides the following predefined data type to hold RPC communications status information:

**error_status_t**

The values that can be contained in the **error_status_t** data type are compatible with the **unsigned long** and **unsigned32** IDL data types. These data types are used for status values in DCE. The **error_status_t** data type contains an additional semantic to indicate that this particular **unsigned long** contains a DCE format error status value. This additional semantic enables the IDL compiler to perform any necessary translation when moving the status value between systems with differing hardware architectures and software operating systems. If you are using status codes that are not in the DCE error status format, or if you do not require such conversion, use an **unsigned long** instead of **error_status_t**.

## International Characters

The implicitly imported file **nbase.idl** provides predefined data types to support present and emerging international standards for the representation of characters and strings:

**ISO_LATIN_1**
**ISO_MULTI_LINGUAL**
**ISO_UCS**

Data of type **char** is subject to ASCII-EBCDIC conversion when transmitted by the RPC mechanism. The predefined international character types are constructed from the base type **byte** and are thereby protected from data representation format conversion.

The **ISO_LATIN_1** type is represented in 8 bits and is predefined as follows:

```
typedef byte ISO_LATIN_1;
```

The **ISO_MULTI_LINGUAL** type is represented in 16 bits and is predefined as follows:

```
typedef struct {
     byte row, column;
     } ISO_MULTI_LINGUAL;
```

The **ISO_UCS** type is represented in 32 bits and is predefined as follows:

```
typedef struct {
     byte group, plane, row, column;
     } ISO_UCS;
```

## Constructed Data Types

The following subsections describe the constructed data types that are provided by IDL. The constructed types are built on the basic data types, which are described in “Basic Data Types” on page 237. The constructed data types are:

- Structures
- Unions
- Enumerations
- Pipes
- Arrays
- Strings

In IDL, as in C, arrays and pointers are specified using declarator constructs. The other constructed types are specified using type specifiers.

## Structures

The *type_specifier* for a structure type can take the following forms:

**struct** *[tag]*
 **{**
  *struct_member*;
  ...
 **}**

**struct** *tag*

A *tag*, if supplied in a specifier of the first form, becomes a shorthand form for the set of member declarations that follows it. Such a *tag* can subsequently be used in a specifier of the second form.

Note that if the structure data type includes nested structures, each structure must be identified with a unique tag, as follows:.

```
struct tag1
 {
  struct_member1;
  ...
     struct tag2
      {
       struct_member2;
       ...
      }
 }
```

A *struct_member* takes the following form:

*[***[***struct_member_attribute***, ...]***] type_specifier declarator*, ...**;**

A *struct_member_attribute* can be any of the following:

| | |
|---|---|
| *array_attribute* | One of several attributes that specify characteristics of arrays. |
| **ignore** | An attribute indicating that the pointer member being declared is not to be transmitted in RPCs. |
| **ref** | An attribute indicating that the pointer member being declared is a reference pointer: it cannot be **NULL** and cannot be an alias. |
| **ptr** | An attribute indicating that the pointer member being declared is a full pointer: it can be **NULL** and can be an alias. |
| **unique** | An attribute indicating that the pointer member being declared is a unique pointer; it can be **NULL** but cannot be an alias. |
| **string** | An attribute indicating that the array member being declared is a string. |

A structure can contain a conformant array only as its last member. Such a structure can be contained by another structure only as its last member. This requirement iterates through any other embedding structures. A structure that contains a conformant array (a *conformant structure*) cannot be returned by an operation as its value and cannot be simply an **out** parameter.

A structure cannot contain a pipe or context handle.

The **ignore** attribute specifies that the pointer is not to be transmitted in RPCs. The **ignore** attribute can be applied only to a pointer that is a member of a structure. The **ignore** attribute is not allowed in a type declaration that defines a pointer type.

## Unions

IDL provides two types of unions: encapsulated and nonencapsulated. An IDL union must be discriminated. In an encapsulated union, the discriminator is part of the union. In a nonencapsulated union, the discriminator is not part of the union.

The following *type_specifier* can be used to indicate either kind of union.

**union** *[tag]*

A definition of the union identified by *tag* must appear elsewhere in the interface definition.

**Encapsulated Unions:** To define an encapsulated union, use the following syntax:

**union** *[tag]* **switch** *(disc_type_spec discriminator)***[union_name**]
**{**
 *case*
 ...
 *[default_case]*
 **}**

If a *tag* is supplied, it can be used in a *type_specifier* of the form shown in the preceding section describing unions.

The *disc_type_spec* indicates the type of the *discriminator*, which can be an integer, a character, a **boolean**, or an enumeration.

The *union_name* specifies a name to be used in C code generated by the IDL compiler. When the IDL compiler generates C code to represent an IDL union, it embeds the union and its discriminator in a C structure. The name of the IDL union becomes the name of the C structure. If you supply a *union_name* in your type declaration, the compiler assigns this name to the embedded C union; otherwise, the compiler assigns the generic name **tagged_union**.

A *case* contains one or more labels and may contain a member declaration:

**case** *constant***:**
...
*[union_member]***;**

Each label in a *case* specifies a constant. The *constant* can take any of the forms accepted in an integer, character, or Boolean constant declaration, each of which is described earlier in this chapter.

A *default_case* can be coded anywhere in the list of cases:

**default:**
*[union_member]***;**

A *union_member* takes the following form:

*[*[*union_member_attribute***,** ...]*] type_specifier declarator***;**

A *union_member_attribute* can be any of the following:

**ptr** An attribute indicating that the pointer member being declared is a full pointer: it can be **NULL** and can be an alias.

**string** An attribute indicating that the character array member being declared is a string.

In any union, the type of the discriminator and the type of all constants in all case labels must resolve to the same type. At the time the union is used, the value of the discriminator selects a member, as follows:

- If the value of the discriminator matches the constant in any label, the member associated with the label is selected.
- If there is no label whose constant matches the value of the discriminator and there is a default case, the default member is selected.

- If there is no label whose constant matches the value of the discriminator and there is no default case, no member is selected and the exception **rpc_x_invalid_tag** is raised.

Note that IDL prohibits duplicate constant label values.

A *union_member* can contain only one declarator. If no *union_member* is supplied, the member is **NULL**; if that member is selected when the union is used, no data is passed. Note, however, that the tag is always passed.

A union cannot contain a pipe, a conformant array, a varying array, or any structure that contains a conformant or varying array. A union also cannot contain a **ref** or **unique** pointer or any structure that contains a **ref** or **unique** pointer.

The following is an example of an encapsulated union.

```
/* IDL construct /*

   typedef
       union fred switch (long a) ralph {
            case 1: float b;
            case 2: long c;
       } bill;

/* becomes in the generated header file /*

  typedef
      struct fred {
          long a;
          union {
              float b;
              long c;
          } ralph;
      } bill;
```

**Nonencapsulated Unions:** To define a nonencapsulated union, use the following syntax:

*[switch_type(datatype)]* **union** *[tag]*
**{**
*case*
...
*[default_case]*
**}**

If a tag is supplied, it can be used in a *type_specifier* of the form shown in the section describing unions. The rules for *case* and *default_case* are the same as those for encapsulated unions.

A parameter or a structure field that is a nonencapsulated union must have an attribute attached to it. This attribute has the form:

*switch_is*(*attr_var*)

The *attr_var* is the name of the parameter or structure field that is the discriminator for the union.

If a nonencapsulated union is used as a structure field, the discriminator of the union must be a field of the same structure. If a nonencapsulated union is used as a parameter of an operation, the discriminator must be another parameter of the same operation.

The following example shows uses of a nonencapsulated union.

```
typedef
  [switch_type(long)] union {
    [case (1,3)] float a_float;
    [case (2)] short b_short;
    [default] ; /* An empty arm. Nothing is shipped. */
  } n_e_union_t;

typedef
  struct {
    long a; /* The discriminant for the union later in this struct. */
    [switch_is (a)] n_e_union_t b;
  } a_struct;

 /* Note switch can follow union in operation */
void op1 (
     [in] handle_t h,
     [in,switch_is (s)] n_e_union_t u,
     [in] long s  );
```

## Enumerations

An IDL enumeration provides names for integers. It is specified as follows:

**enum {***identifier*, ...**}**

Each *identifier* in an enumeration is assigned an integer value. The first identifier is assigned 0 (zero), the second is assigned 1, and so on.

An enumeration can have up to 32,767 identifiers.

## Pipes

IDL supports pipes as a mechanism for transferring large quantities of typed data. An IDL pipe is an open-ended sequence of elements of one type. A pipe permits application-level optimization of bulk data transfer by allowing the overlap of communication and processing. Applications that process a stream of data as it arrives rather than storing the data in memory can make efficient use of the pipe mechanism.

A pipe is specified as follows:

**pipe** *type_specifier*

The *type_specifier* specifies the type for the elements of the pipe. This type cannot be a pointer type, a type that contains a pointer, a conformant type, a context handle, a **handle_t** element type, or a data type that is declared as **transmit_as**.

A pipe type can be used to declare only the type of an operation parameter. IDL recognizes three kinds of pipes, based on the three operation parameters:

- An **in** pipe is used for transferring data from a client to a server. It allows the callee (server) to *pull* an open-ended stream of typed data from the caller (client).
- An **out** pipe is used for transferring data from a server to a client. It allows the callee (server) to *push* the stream of data to the caller (client).
- An **in,out** pipe provides for two-way data transfer between a client and server by combining the behavior of **in** and **out** pipes.

A pipe can be defined only through a **typedef** declaration. Anonymous pipe types are not supported.

At the interface between the stub and the application-specific code (for both the client and server), a pipe appears as a simple callback mechanism. To the user code, the processing of a pipe parameter appears to be synchronous. The IDL implementation of pipes in the RPC stub and runtime allows the apparent callbacks to occur without requiring actual remote callbacks. As a result, pipes provide an efficient transfer mechanism for large amounts of data.

Note that pipe data communications occur at the same speed as arrays. Pipes can improve latency and minimize memory utilization, but not throughput. Pipes are intended for use where the receiver can process the data in some way as it arrives, for example, by writing it to a file or passing it to a consumer thread. If the intent is to store the data in memory for later processing, pipes offer no advantage over arrays.

**Note:** Control characters on different platforms may be interpreted differently by their editors. When using pipes to transfer data or files, consider the potential affect on the data.

**IDL Pipes Example:** To illustrate the IDL implementation of pipes, consider the following IDL fragment:

```
typedef
   pipe element_t pipe_t;
```

When the code containing this fragment is compiled, the IDL compiler will generate the following declarations in the derived header file:

```
typedef struct pipe_t {
void (* pull)(
#ifdef IDL_PROTOTYPES
rpc_ss_pipe_state_t state,
element_t *buf,
idl_ulong_int esize,
idl_ulong_int *ecount
#endif
);
void (* push)(
#ifdef IDL_PROTOTYPES
rpc_ss_pipe_state_t state,
element_t *buf,
idl_ulong_int  ecount
#endif
);
void (* alloc)(
#ifdef IDL_PROTOTYPES
rpc_ss_pipe_state_t state,
idl_ulong_int bsize,
element_t **buf,
idl_ulong_int *bcount
#endif
);
rpc_ss_pipe_state_t state;
} pipe_t;
```

The pipe data structure specifies pointers to three separate routines and a pipe state. The client application has to implement these routines for the client stub to call, and the server manager must call the associated routines generated in the server stub.

The *pull* routine is used for an input pipe. It pulls the next chunk of data from the client application into the pipe. The input parameters include the pipe **state**, the buffer (***buf**) containing a chunk of data, and the size of the buffer (**esize**) in terms of the number of pipe data elements. The output parameter is the actual count (***ecount**) of the number of pipe data elements in the buffer.

The *push* routine is used for an output pipe. It pushes the next chunk of data from the pipe to the client application. The input parameters include the pipe **state**, the buffer (***buf**) containing a chunk of data, and a count (**ecount**) of the number of pipe data elements in the buffer.

The *alloc* routine allocates a buffer for the pipe data. The input parameters include the pipe **state** and the requested size of the buffer (**bsize**) in bytes. The output parameters include a pointer to the allocated buffer (****buf**), and the actual count (**bcount**) of the number of bytes in the buffer. The routine allocates memory from which pipe data can be marshalled or into which pipe data can be marshalled. If less memory is allocated than requested, the RPC runtime uses the smaller memory and makes more callbacks to the user. If the routine allocates more memory than requested, the excess memory is not used.

Finally, the *state* is used to coordinate between the above routines.

For more on how to write the code for the client and server manager, see "Pipes" on page 192.

**Rules for Using Pipes:** Observe the following rules when defining pipes in IDL:

- Pipe types must only be parameters. In other words, pipes of pipes, arrays of pipes, and structures or unions containing pipes as members are not permitted.
- A pipe cannot be a function result.
- The element type of a pipe cannot be a pointer or contain a pointer.
- The element type of a pipe cannot be a **context_handle** or **handle_t** type.
- A pipe type cannot be used in the definition of another type. For example, the following code fragment is not permitted:

  ```
  typedef
     pipe char pipe_t;

  typedef
     pipe_t * pipe_p_t;
  ```

- A pipe type cannot have the **transmit_as** attribute.
- The element type of a pipe cannot have the **transmit_as** attribute.
- A pipe parameter can be passed by value or by reference. A pipe that is passed by reference (that is, has an * (asterisk)) cannot have the **ptr** or **unique** parameter attributes.
- Pipes that pass data from the client to the server must be processed in the order in which they occur in an operation's signature. All such pipes must be processed before data is sent from the server to the client.
- Pipes that pass data from the server to the client must be processed in the order in which they occur in an operation's signature. No such pipes must be processed until all data has been sent from the client to the server.
- Manager routines must reraise RPC pipe and communication exceptions so that client stub code and server stub code continue to run properly.

  For example, consider an interface that has an **out** pipe along with other **out** parameters. Suppose that the following sequence of events occurs:

  – The manager routine closes the pipe by writing an empty chunk whose length is 0 (zero).
  – The manager routine attempts to write another chunk of data to the pipe.
  – The generated **push** routine raises the exception **rpc_x_fault_pipe_closed**.
  – The manager routine catches the exception and does not reraise it.
  – The manager routine exits normally.

  – The server stub attempts to marshall the **out** parameters.

  After this sequence, neither the server stub nor the client stub can continue to run properly. To avoid this situation, you *must* reraise the exception.

- A pipe cannot be used in an operation that has the **broadcast** or **idempotent** attribute.
- The element type of a pipe cannot be a conformant structure.
- The maximum length of pipe type IDs is 29 characters.

## Arrays

IDL supports the following types of arrays:

- Fixed: The size of the array is defined in the IDL and all of the data in the array is transferred during the call.
- Conformant: The size of the array is determined at run time by the value of the field or parameter referred by a **min_is** or **max_is** or **size_is** attribute. All of the data in the array is transferred during the call.

  **Note:** For **[in, out]** conformant character arrays, if the **min_is** or **max_is** or **size_is** attribute is not specified, the array size is implicitly known by the client runtime. A change to a larger array size by the server side of the RPC call could result in client failure, because of a memory violation. To avoid this possibility, you should use the **min_is** or **max_is** or **size_is** attribute when specifying conformant arrays.
- Varying: The size of the array is defined in the IDL, but the part of its contents transferred during the call is determined by the values of fields or parameters named in one or more data limit attributes. The data limit attributes are **first_is**, **length_is**, and **last_is**.

An array can also be both conformant and varying (or, as it is sometimes termed, *open*). In this case, the size of the array is determined at run time, by the value of the field or parameter referred by the **min_is** or **max_is** or **size_is** attribute. The part of its contents transferred during the call is determined by the values of fields or parameters named in one or more of the data limit attributes.

An IDL array is declared using an *array_declarator* construct whose syntax is as follows:

  *array_identifier array_bounds_declarator* ...

An *array_bounds_declarator* must be specified for each dimension of an array.

**Array Bounds:** The *array_bounds_declarator* for the first dimension of an array can take any of the following forms:

| | |
|---|---|
| **[***lower* **..** *upper***]** | The lower bound is *lower*. The upper bound is *upper*. |
| **[***size***]** | The lower bound is 0 (zero). The upper bound is *size* - 1. |
| **[* ]** | The lower bound is 0 (zero). The upper bound is determined by a **max_is** or **size_is** attribute. |
| **[ ]** | Same as the preceding explanation. |
| **[***lower* **.. *]** | The lower bound is *lower*. The upper bound is determined by a **max_is** or **size_is** attribute. |
| **[ * ..** *upper***]** | The lower bound is determined by a **min_is** attribute. The upper bound is upper. |
| **[ * .. * ]** | The lower bound is determined by a **min_is** attribute. The upper bound is determined by a **size_is** or **max_is** attribute. |

**Conformance in Dimensions Other Than the First:** If a multidimensional array is conformant in a dimension other than the first, the C description for this array, which is located in the header file generated by the IDL compiler, will be a one-dimensional conformant array of the appropriate element type. This occurs because there is no natural C binding for conformance in dimensions other than the first.

The following examples show how IDL type definitions and parameter declarations that contain bounds in dimensions other than the first are translated into their C equivalents at run time.

**IDL Type Definition:**

```
typedef struct {
    long a;
    long e;
    [max_is(,,e),min_is(a)] long g7[a..1][2..9][3..e];
} t3;
```

**C Translation:**

```
typedef struct  {
  idl_long_int a;
  idl_long_int e;
  idl_long_int g7[1];
} t3;
```

**IDL Parameter Declaration:**

```
[in,out,max_is(,,e),min_is(a)] long g7[*..1][2..9][3..*];
```

**C Translation:**

```
/* [in, out] */ idl_long_int g7[];
```

Arrays that have a nonzero first lower bound and a first upper bound that is determined at run time are translated into the equivalent C representation of a conformant array, as shown in the following IDL type definition and parameter declaration examples:

**IDL Type Definition:**

```
typedef struct  {
           long s;
           [size_is(s)] long fa3[3..*][-4..1][-1..2];
} t1;
```

**C Translation:**

```
typedef struct  {
  idl_long_int s;
  idl_long_int fa3[1][6][4];
} t1;
```

**IDL Parameter Declaration:**

```
[in,out,size_is(s)] long fa3[3..*][-4..1][-1..2];
```

**C Translation:**

```
/* [in, out] */ idl_long_int fa3[][6][4];
```

**Array Attributes:** Array attributes specify the size of an array or the part of an array that is to be transferred during a call. An array attribute specifies a variable that is either a field in the same structure as the array or a parameter in the same operation as the array.

An *array_attribute* can take the following forms:

**min_is (***[*] variable***)**
**max_is (***[*] variable***)**
**size_is (***[*] variable***)**
**last_is (***[*] variable***)**
**first_is (***[*] variable***)**
**length_is (***[*] variable***)**

A *variable* specifies a variable whose value at run time will determine the bound or element count for the associated dimension. A pointer variable is indicated by preceding the variable name with an ***** (asterisk).

If the array is a member of a structure, any referred variables must be members of the same structure. If the array is a parameter of an operation, any referred variables must be parameters of the same operation.

Only the **..._is(***variable***)** form is allowed when the array is a field of a structure. In this case, the **..._is(****variable* **)** form is not allowed. An array with an array attribute (that is, a conformant or varying array) cannot have the **transmit_as** attribute.

**The min_is Attribute**

The **min_is** attribute is used to specify the variable(s) from which the values of one or more lower bounds of the array will be obtained at run time. If any dimension of an array has an unspecified lower bound, the array must have a **min_is** attribute. A variable must be identified for each such dimension. The following examples show the syntax of the **min_is** attribute:

```
/* Assume values of variables are as follows
    long a = -10;
    long b = -20;
    long c = -30;
*/

long [min_is(a)] g1[*..10];           /* g1[-10..10] */
long [min_is(a)] g2[*..10][4];        /* g2[-10..10[0..3] */
long [min_is(a,b)] g3[*..10][*..20];  /* g3[-10..10][-20..20] */
long [min_is(,b)] g4[2][*..20];       /* g4[0..1][-20..20] */
long [min_is(a,,c)] g5[*..7][2..9][*..8];   /* g5[-10..7][2..9][-30..8] */
long [min_is(a,b,)] g6[*..10][*..20][3..8]; /* g6[-10..10][-20..20][3..8] */
```

The following examples show the **min_is** attribute being applied to the first dimension of an array in an IDL type definition and parameter declaration, and how the definition or parameter is translated into its C equivalent:

**IDL Type Definition:**

```
     typedef struct {
             long n;
       [min_is(n)] long fa3[*..10][-4..1][-1..2]
     } t2;
```

**C Translation:**

```
     typedef struct  {
       idl_long_int n;
       idl_long_int fa3[1][6][4];
```

```
    } t2;
```

**IDL Parameter Declaration:**

```
    [in,out,min_is(n)] long fa3[*..10][-4..1][-1..2];
```

**C Translation:**

```
    /* [in, out] */ idl_long_int fa3[][6][4];
```

### The max_is Attribute

The **max_is** attribute is used to specify the variables from which the values of one or more upper bounds of the array are obtained at run time. If any dimension of an array has an unspecified upper bound, the array must have a **max_is** or **size_is** attribute. A variable must be identified for each dimension in which the upper bound is unspecified. In a **max_is** attribute, the value in the identified variable specifies the maximum array index in that dimension. An array with one or more unspecified upper bounds may have a **max_is** attribute or a **size_is** attribute, but not both.

The **max_is** attribute is for use with conformant arrays. Following are some examples of the **max_is** attribute:

```
/* Assume values of variables are as follows:
    long a = 10;
    long b = 20;
    long c = 30;
*/

long [max_is(a)] f1[];           /* f1[0..10] /*
long [max_is(a)] f2[][4];        /* f2[0..10][0..3]  */
long [max_is(a,b)] f3[][];       /* f3[0..10][0..20] */
long [max_is(,b)] f4[2][];       /* f4[0..1][0..20] */
long [max_is(a,,c)] f5[1..*][2..9][3..*];  /* f5[1..10][2..9][3..30] */
long [max_is(a,b,)] f6[1..*][2..*][3..8];  /* f6[1..10][2..20][3..8] */
```

### The size_is Attribute

The **size_is** attribute is used to specify the variables from which the values of the element counts for one or more dimensions of the array are obtained at run time. If any dimension of an array has an unspecified upper bound, the array must have a **max_is** or **size_is** attribute. A variable must be identified for each dimension in which the upper bound is unspecified. In a **size_is** attribute, the value in the identified variable specifies the number of elements in that dimension. An array with one or more unspecified upper bounds may have a **max_is** attribute or a **size_is** attribute, but not both.

The size of a dimension is defined as the upper bound, minus the lower bound, plus one.

The **size_is** attribute is for use with conformant arrays. Following is an example of the **size_is** attribute:

```
/* Assume the following values for the referenced variables:
   n3 = 5;
   x2 = 12;
   x3 = 14;
   z2 = 9;
   z3 = 10;
*/

/* The following declaration */

int [min_is(,,n3),max_is(,x2,x3)] hh[3..13,4..*,*..*];

/* specifies the same data to be transmitted as the declaration */

int [min_is(,,n3),size_is(,z2,z3)] hh[3..13,4..*,*..*];
```

### The last_is Attribute

The **last_is** attribute is one of the attributes that can be used to allow the amount of data in an array that will be transmitted to be determined at run time. Each **last_is** attribute specifies an upper data limit, which is the highest index value in that dimension for the array elements to be transmitted. If the entry in a **last_is** attribute for a dimension is empty, the effect is as if the upper bound in that dimension had been specified.

An array can have either the **last_is** attribute or the **length_is** attribute, but not both.

When an array with the **last_is** attribute is used in a remote procedure call, the elements actually passed in the call can be a subset of the maximum possible.

The **last_is** attribute is for use with varying arrays. Following is an example of the **last_is** attribute:

```
/* Assume the following values for the referenced variables:
    long a = 1;
    long b = 2;
    long c = 3;
    long e = 25;
    long f = 35;
*/

long [last_is(a,b)] bb1[10][20];   /* transmit bb1[0..1][0..2] */
long [last_is(a,b)] bb2[-1..10][-2..20][-3..30];
                                   /* transmit bb2[-1..1][-2..2][-3..30] */
long [last_is(a,,c)] bb3[-1..10][-2..20][-3..30];
                                   /* transmit bb3[-1..1][-2..20][-3..3] */
long [last_is(,b,c),max_is(,e)] cc1[10][][30];
                                   /* transmit cc1[0..9][0..2][0..3] */
long [last_is(a,b),max_is(,e,f)] cc2[-4..4][][];
                                   /* transmit cc2[-4..1][0..2][0..35] */
```

### The first_is Attribute

The **first_is** attribute is one of the attributes that can be used to allow the amount of data in an array that will be transmitted to be determined at run time. Each **first_is** attribute specifies a lower data limit, which is the lowest index value in that dimension for the array elements to be transmitted. If the entry in a **first_is** attribute for a dimension is empty, the effect is as if the lower bound in that dimension had been specified.

When an array with the **first_is** attribute is used in a remote procedure call, the elements actually passed in the call can be a subset of the maximum possible.

The **first_is** attribute is for use with varying arrays. Following is an example of the **first_is** attribute:

```
/* Assume the following values for the referenced variables:
    long p = -1;
    long q = -2;
    long r = -3;
    long t = -25;
    long u = -35;
    long x = 1;
    long y = 2;
    long z = 3;
*/

long [first_is(p)] dd1[-10..10];   /* transmit dd1[-1..10] */
long [first_is(p),last_is(x)] dd2[-10..10];  /* transmit dd2[-1..1] */
long [first_is(p,q)] ee1[-10..10][-20..20];  /* transmit ee1[-1..10][-2..20] */
long [first_is(p,q)] ee2[-10..10][-20..20][-30..30];
```

```
                                /* transmit ee2[-1..10][-2..20][-30..30] */
long [first_is(p,q,r),last_is(,,z)] ee3[-10..10][-20..20][-30..30];
                                /* transmit ee3[-10..10][-20..20][-30..30] */
double [first_is(,q,r),min_is(,t)] ff1[10][*..2][-30..30];
                                /* transmit ff1[0..9][-2..2][-3..30] */
double [first_is(p,q),min_is(,t,u)] ff2[-4..4][*..2][*..35];
                                /* transmit ff2[-1..4][-2..2][-35..35] */
double [max_is(x,,z),min_is(,t,u),first_is(p,,r)] ff3[-20..*][*..30][*..*]
                                /* transmit ff3[-1..1][-25..30][-3..3] */
```

**The length_is Attribute**

The **length_is** attribute is one of the attributes that can be used to allow the amount of data in an array that will be transmitted to be determined at run time. Each **length_is** attribute specifies the number of elements in that dimension to be transmitted. If the entry in a **length_is** attribute for a dimension is empty, the effect is for the highest index value in that dimension for the elements to be transmitted to be determined from the upper bound in that dimension.

An array can have either the **last_is** attribute or the **length_is** attribute, but not both.

When an array with the **length_is** attribute is used in an RPC, the elements actually passed in the call can be a subset of the maximum possible.

The **length_is** attribute is for use with varying arrays. Following is an example of the **length_is** attribute:

```
/* Assume the following values for the referenced variables:
    n3 = 5;
    f2 = 10;
    a1 = 11;
    a2 = 12;
    a3 = 14;
    e1 = 9;
    e2 = 3;
    e3 = 10;
*/

/* The following declaration: */

int [min_is(,,n3),first_is(,f2,),last_is(a1,a2,a3)] gg[3..13,4..14,*..15];

/* specifies the same data to be transmitted as the declaration: */

int [min_is(,,n3),first_is(,f2,),length_is(e1,e2,e3)] gg[3..13,4..14,*..15];
```

**Rules for Using Arrays:** Observe the following rules when defining arrays in IDL:

- A structure can contain only one conformant array, which must be the last member in the structure.
- Conformant arrays are not valid in unions.
- A structure containing a conformant array can be passed only by reference.
- Arrays that have the **transmit_as** attribute cannot be conformant or varying arrays.
- The structure member or parameter referred in an array attribute cannot be defined to have either the **represent_as** or **transmit_as** attribute.
- Array bounds must be integers. Array attributes can only refer to structure members or parameters of integer type.
- A parameter that is referred by an array attribute on a conformant array must have the **in** attribute.
- Array elements cannot be context handles or pipes, or conformant arrays or conformant structures.

## Strings

IDL uses strings as one-dimensional arrays to which the **string** attribute is assigned. The element type of the array must resolve to one of the following:

- Type **char**
- Type **byte**
- A structure all of whose members are of type **byte** or of a named type that resolves to **byte**
- A named type that resolves to one of the three previous types
- Type **unsigned short**
- Type **unsigned long**
- A named type that resolves to **unsigned short** or **unsigned long**

  Strings built from **byte** or **char** data types are referred to as byte-string types while strings built from **unsigned short** or **unsigned long** types are called integer-string types. Integer-string types allow for multi-octet character sets whose characters are represented by 16-bit or 32-bit quantities, rather than as groups of bytes, for example:

```
/* A structure that contains a fixed string */
/* and a conformant string */
typedef unsigned long PRIVATE_CHAR_32;
typedef struct {
   [string] PRIVATE_CHAR_32 fixed[27];
   [string] PRIVATE_CHAR_32 conf[];
} two_strings;
```

```
/* A structure that contains pointers to two strings */
typedef unsigned short PRIVATE_CHAR_16;
typedef struct {
   [string] PRIVATE_CHAR_16 *astring;
   [string] PRIVATE_CHAR_16 *bstring;
} stringptrs;
```

  Integer-string types use the array element zero (0) to specify the string terminator, while byte-string types use the NULL character. Both byte-type and integer-type strings conform to the same usage rules (described below).

An array with the **string** attribute represents a string of characters. The **string** attribute does not specify the format of the string or the mechanism for determining its length. Implementations of IDL provide string formats and mechanisms for determining string lengths that are compatible with the programming languages in which applications are written. For DCE RPC IDL, the number of characters in a **string** array includes the **NULL** terminator (for byte-string types) or the zero (0) terminator (for integer-string types), and the entire string is passed between stubs.

The *array_bounds_declarator* for a **string** array determines the maximum number of characters in the array. When you declare a string, you must allocate space for one more than the maximum number of characters the string is to hold. For instance, if a string is to store 80 characters, the string must be declared with a size of 81:

```
/* A string type that holds 80 characters */
typedef
   [string] char string_t [81];
```

If an array has the **string** attribute or if the type of an array has the **string** attribute, the array cannot have the **first_is**, the **last_is**, or the **length_is** attribute.

## Pointers

Use the following syntax to declare an IDL pointer:

**[*]...pointer_identifier*

The * (asterisk) is the pointer operator, and multiple asterisks indicate multiple levels of indirection.

**Pointer Attributes:** Pointers are used for several purposes. This includes carrying out a parameter passing mechanism that allows a data value to be returned and building complex data structures. IDL offers three classes of pointers: reference pointers, full pointers, and unique pointers. The attributes that indicate these pointers are:

- **ref**: Indicates reference pointers. This is the default for top-level pointers used in parameters.
- **ptr**: Indicates full pointers.
- **unique**: Indicates unique pointers.

Pointer attributes are used in parameters, structure and union members, and in type definitions. In some instances, IDL infers the applicable pointer class from its usage. However, most pointer declarations require that you specify a pointer class by using one of the following methods:

- Use the **ref**, **ptr**, or **unique** attribute in the pointer declaration.
- Use the **pointer_default** attribute in the IDL interface heading. The default pointer class is determined by the **pointer_default** attribute.

Pointer attributes are applied only to the top-level pointer within the declaration. If multiple pointers are declared in a single declaration, the **pointer_default** established applies to all but the top-level pointer. (See "Pointer Attributes in Parameters" on page 257.)

Examples of pointers are shown later in this section.

**Reference Pointers**

A reference pointer is the less complex form of pointer. The most common use for this class of pointer is as a passing mechanism, for example, passing an integer by reference. Reference pointers have significantly better performance than full pointers, but are restrictive. You cannot create a linked list using a reference pointer because a reference pointer cannot have a **NULL** value, and the list cannot be ended.

A reference pointer has the following characteristics:

- It always points to valid storage; it can never have a **NULL** value.
- Its value does not change during a call; it always points to the same storage on return from the call as it did when the call was made.
- It does not support aliasing; it cannot point to a storage area that is pointed to by any other pointer used in a parameter of the same operation.

When a manager routine is entered, all the reference pointers in its parameters will point to valid storage, except those reference pointers that point neither to targets whose size can be determined at compile time nor to values that have been received from the client.

In the following example, the size of the targets of the reference pointers can be calculated at compilation time:

```
typedef [ref] long *rpl;

void op1( [in] long f,
          [in] long l,
          [in,first_is(f),last_is(l)] rpl rpla[10] );
```

For this example, when the manager is entered, all the pointers in `rpla` will point to usable storage, although only `*rpla[f]` through `*rpla[l]` will be the values received from the client.

Conversely, the size of the targets of the reference pointers cannot be calculated at compile time in the following example:

```
typedef [ref,string] char *rps;

void op1( [in] long f,
          [in] long l,
          [in,first_is(f),last_is(l)] rps rpsa[10] );
```

In this case, only `rpsa[f]` through `rpsa[l]`, which point to values received from the client, will point to usable storage.

### Full Pointers

A full pointer is the more complex form of pointer. It supports all capabilities associated with pointers. For example, by using a full pointer, you can build complex data structures such as linked lists, trees, queues, or arbitrary graphs.

A full pointer has the following characteristics:

- Its value can change during a call; it can change from a **NULL** to non-**NULL** value, non-**NULL** to **NULL**, or from one non-**NULL** value to another non-**NULL** value.
- It supports aliasing; it can point to a storage area that is also pointed to by any other full pointer used in a parameter of the same operation. However, all such pointers must point to the beginning of the structure. There is no support for pointers to substructures or to overlapping storage areas. For example, if the interface definition code contains the following:

  ```
  [uuid(0e256080-587c-11ca-878c-08002b111685), version(1.0)]
  interface overlap
  {
    typedef struct {
            long bill;
            long charlie;
    } foo;
    typedef struct {
            long fred;
            foo ken;
    } bar;

    void op ( [in] foo *f, [in] bar *b );
  }
  ```

  The client application code includes:

  ```
  bar bb;
   .
   .
  op ( &bb.ken, &bb );
  ```

  The server stub treats these two separate parameters as distinct, and the manager application code does not see them as overlapping storage.
- It allows dynamically allocated data to be returned from a call.

**Unique Pointers**

A unique pointer is more flexible than a reference pointer. However, both types of pointers share several important characteristics.

A unique pointer has the following characteristics:

- It can have a **NULL** value.
- It can change from **NULL** to non-**NULL** during a call. This change results in memory being allocated on return from the call, whereby the result is stored in the allocated memory.
- It can change from non-**NULL** to **NULL** during a call. This change can result in the orphaning of the memory pointed to on return from the call. Note that if a unique pointer changes from one non-**NULL** value to another non-**NULL** value, the change is ignored.
- It does not identify particular extents of memory, but only extents of memory that are suitable for storing the data. If it is important to know that the data is being stored in a specific memory location, then you should use a full pointer.
- If it has a value other than **NULL**, output data is placed in existing storage.

Unique pointers are similar to reference pointers in the following ways:

- No storage pointed to by a unique pointer can be reached from any other name in the operation. That is, a unique pointer does not allow aliasing of data within the operation.
- Data returned from the called subroutine is written into the existing storage specified by the unique pointer, if the pointer did not have the value **NULL**.

With regard to performance, unique pointers have an advantage over full pointers because unique pointers do not support the referencing of common data by more than one pointer (aliasing), and they are significantly more flexible than reference pointers because they can have a value of **NULL**.

Unique pointers are particularly suitable for creating optional parameters (because you can specify them as **NULL**) and for simple tree or singly linked-list data structures. You specify the three different levels of pointers by attributes, as follows:

| | |
|---|---|
| **[ref]** | Reference pointers |
| **[unique]** | Unique pointers |
| **[ptr]** | Full pointers |

The following example shows how a unique pointer can be used:

```
[
   uuid(d37a0e80-5d23-11c9-b199-08002b13d56d)
] interface Unique_ptrs
{
   typedef [ref]    long *r_ptr;
   typedef [unique] long *u_ptr;
   typedef [ptr]    long *f_ptr;

   void op1 (
     [ref,in,out,string]    char *my_rname,
     [unique,in,out,string]   char *my_uname,
     [ptr,in,out,string]    char *my_pname
            );
 }
```

**Pointer Attributes in Parameters:** A pointer attribute can be applied to a parameter only if the parameter contains an explicit pointer declaration (*****).

By default, a single pointer (*****) operator in a parameter list of an operation declaration is treated as a reference pointer. To override this, specify a pointer attribute for the parameter.

When there is more than one pointer operator, or multiple levels of indirection in the parameter list, the rightmost pointer is the top-level pointer; all pointers to the left of the rightmost pointer are of a lower level. The top-level pointer is treated as a reference pointer by default; the lower level pointers have the semantics specified by the **pointer_default** attribute in the interface.

The following example illustrates the use of top- and lower-level pointers:

```
void op1 ([in] long **p_p_1);
```

In this example, p_p_1 is a pointer to a pointer to a long integer. The first or leftmost pointer (*) signifies that the pointer to the long is a lower-level pointer, and the second or rightmost pointer (*) signifies that the pointer to the pointer is a top-level pointer.

Any pointer attribute you specify for the parameter applies to the top-level pointer only. Note that unless you specify a pointer attribute, the top-level explicit pointer declaration in a parameter defaults to a reference pointer even if the **pointer_default(ptr)** interface attribute is specified.

Using a reference pointer improves performance but is more restrictive. For example, the pointer declared in the following operation, for the parameter **int_value**, is a reference pointer. An application call to this operation can never specify **NULL** as the value of **int_value**.

```
void op ([in] long *int_value);
```

To pass a **NULL** value, use a full pointer. The following two methods make **int_value** into a full pointer:

- Applying the **ptr** attribute to the declaration of the parameter, **int_value**

  ```
  void op ([in, ptr] long *int_value);
  ```

- Using the **pointer_default (ptr)** attribute in an interface header

  ```
  [uuid(135e7f00-1682-11ca-bf61-08002b111685,
   pointer_default(ptr),
   version(1.0)] interface full_pointer
  {
  typedef long *long_ptr;
  void op ([in] long_ptr int_value);
  }
  ```

**Array Attributes on Pointers:** To apply array attributes to pointers, use the **[max_is]** or **[size_is]** attributes. When applied to a pointer, the **[max_is]** and **[size_is]** attributes convert the pointer from a single element of a certain type to a pointer to an array of elements of that type. The number of elements in the array is determined by the variable in the **[max_is]** and **[size_is]** attributes.

**Pointer Attributes in Function Results:** Function results that are pointers are always treated as full pointers. The **ptr** attribute is allowed on function results but it is not mandatory. The **ref** pointer attribute is never allowed on function results.

A function result that is a pointer always indicates new storage. A pointer parameter can refer to storage that was allocated before the function was called, but a function result cannot.

**Pointers in Structure Fields and Union Case:** If a pointer is declared in a member of a structure or union, its default is determined by the **pointer_default** attribute you specify for the interface. To override this, specify a pointer attribute for the member.

**Resolving a Possible Pointer Ambiguity:** A declaration of the following form raises a possible ambiguity about the type of *myarray*:

```
void op([in, out] long x, [in, out, size_is(x)] long **myarray);
```

IDL defines *myarray* in this case to be an array of pointers to **long**s, not a pointer to an array of **long**s. The **max_is** and **size_is** attributes always apply to the top-level, or rightmost, ***** (asterisk) in the IDL signature of a parameter.

**Rules for Using Pointers:** Use the following rules when developing code in IDL:

- Do not use the full pointer attribute on the following:
  - The parameter in the first parameter position, when that parameter is of type **handle_t** or is of a type with the **handle** attribute.
  - Context handle parameters.
  - A parameter that has the output attribute (**out**) only.
- The element type of a pipe must not be a pointer or a structure containing a pointer.
- A member of a union or a structure contained in a union cannot contain a reference pointer.
- A reference pointer must point to valid storage at the time the call is made.
- A parameter containing a varying array of reference pointers must have all array elements initialized to point to valid storage even if only a portion of the array is supplied, because the manager code (the application code supporting an interface on a server) may use the remaining array elements. (Recall that a varying array is one to which any of the array attributes **first_is, last_is, length_is** is applied.)
- The type name in a declaration that defines a pointer type must have no more than 28 characters.

**Memory Management for Pointed-to Nodes:** A full pointer can change its value across a call. Therefore, stubs must be able to manage memory for the pointed-to nodes. Managing memory involves allocating and freeing memory for user data structures.

#### Allocating and Freeing Memory

Manager code within RPC servers usually uses the **rpc_ss_allocate()** routine to allocate storage. Storage that is allocated by **rpc_ss_allocate()** is released by the server stub after any output parameters have been marshalled by the stubs. Storage allocated by other allocators is not released automatically but must be freed by the manager code. When the manager code makes a remote call, the default memory management routines are **rpc_ss_allocate()** and **rpc_ss_free()**.

The syntax of the **rpc_ss_allocate()** routine is as follows:

**idl_void_p_t rpc_ss_allocate (idl_size_t** *size***);**

The *size* parameter specifies the size of the memory allocated. **Note:** In ANSI standard C environments, **idl_void_p_t** is defined as **void ***, and in other environments is defined as **char ***. Use **rpc_ss_free()** to release storage allocated by **rpc_ss_allocate()**. You can also use the **rpc_ss_free()** routine to release storage pointed to by a full pointer in an input parameter and have the free of the memory reflected on return to the calling application by specifying the **reflect_deletions** attribute as an operation attribute. See "Parameter Declarations" on page 236 for more information about declaring IDL operations.

The syntax of the routine is as follows:

**void rpc_ss_free (idl_void_p_t** *node_to_free***);**

The *node_to_free* parameter specifies the location of the memory to be freed.

**Enabling and Disabling Memory Allocation**

It may be necessary to call manager routines from different environments; for example, when the application is both a client and a server of the same interface. In this case, the same routine may be called both from server manager code and from client code. The **rpc_ss_allocate()** routine, when used by the manager code to allocate memory, must be initialized before its first use. The stub performs the initialization automatically. Code, other than stub code, that calls a routine, which in turn calls **rpc_ss_allocate()**, first calls the **rpc_ss_enable_allocate()** routine.

The syntax of the routine is as follows:

**void rpc_ss_enable_allocate (void);**

The environment set up by the **rpc_ss_enable_allocate()** routine is released by calling the **rpc_ss_disable_allocate()** routine. This routine releases all memory allocated by calls to **rpc_ss_allocate()** because the call to **rpc_ss_enable_allocate()** was made. It also releases memory that was used by the memory management mechanism for internal bookkeeping.

The syntax of the routine is as follows:

**void rpc_ss_disable_allocate (void);**

**Advanced Memory Management Support:** Memory management may also involve setting and exchanging the mechanisms used for allocating and freeing memory. The default memory management routines within the manager code are **rpc_ss_allocate()** and **rpc_ss_free()**.

**Setting the Client Memory Mechanism**

Use the **rpc_ss_set_client_alloc_free()** routine to establish the routines used in allocating and freeing memory.

The syntax of the routine is as follows:

```
void rpc_ss_set_client_alloc_free (
    idl_void_p_t (*p_allocate ) (
      idl_size_t size),
     void (*p_free ) (
       idl_void_p_t ptr)
     );
```

The *p_allocate* parameter points to a routine that has the same procedure declaration as the **malloc** routine, used by the client stub when performing memory allocation. The *p_free* parameter points to a routine that has the same procedure declaration as the **free** routine, used by the client stub to free memory.

**Exchanging Client Memory Mechanisms**

This routine exchanges the current client allocation and freeing mechanism for one supplied in the call. The primary purpose of this routine is to simplify the writing of modular routine libraries in which RPCs are made. To preserve modularity, any dynamically allocated memory returned by a modular routine library must be allocated with a specific memory allocator. When dynamically allocated memory is returned by an RPC that is then returned to the user of the routine library, use **rpc_ss_swap_client_alloc_free()** to make sure the desired memory allocator is used. Prior to returning, the modular routine library calls **rpc_ss_set_client_alloc_free()** to restore the previous memory management mechanism.

The syntax of the routine is as follows:

```
void rpc_ss_swap_client_alloc_free (
      idl_void_p_t (*p_allocate ) (
        idl_size_t size),
      void (*p_free ) (
        idl_void_p_t ptr),
      idl_void_p_t (**p_p_old_allocate ) (
        unsigned long size),
      void (**p_p_old_free ) (
        idl_void_p_t ptr)
      );
```

The *p_allocate* parameter points to a routine that has the same procedure declaration as the **malloc** routine, and was used by the client stub when performing memory allocation. The *p_free* parameter points to a routine that has the same procedure declaration as the **free** routine, and was used by the client stub to free memory. The *p_p_old_allocate* parameter points to a pointer to a routine that has the same procedure declaration as the **malloc** routine, and was used for memory allocation in the client stub. The *p_p_old_free* parameter points to a pointer to a routine that has the same procedure declaration as the **free** routine, and was used for memory release in the client.

**Use of Thread Handles in Memory Management:** There are two situations where control of memory management requires the use of thread handles. The more common occurs when the manager thread spawns additional threads. The less common occurs when a program changes from being a client to being a server, and then changes back to a client.

#### Spawning Threads

When an RPC runs the manager code, the manager code may want to spawn additional threads to complete the task for which it was called. To spawn additional threads that can perform memory management, the manager code must first call the **rpc_ss_get_thread_handle()** routine to get its thread handle and then pass that thread handle to each spawned thread. Each spawned thread that uses the **rpc_ss_allocate()** and **rpc_ss_free()** routines for memory management first calls the **rpc_ss_set_thread_handle()** routine by using the handle obtained by the original manager thread.

These routine calls allow the manager and its spawned threads to share a common memory management environment. This common environment enables memory allocated by the spawned threads to be used in returned parameters, and causes all allocations in the common memory management environment to be released when the manager thread returns to the server stub.

The main manager thread must not return control to the server stub before all the threads it spawned are finished running; otherwise, unpredictable results may occur.

The listener thread can cancel the main manager thread if the RPC is orphaned or if a cancelation occurs on the client side of the application. You should code the main manager thread to end any spawned threads before it exits. The code should anticipate exits caused by an unexpected exception or by being canceled.

Your code can handle all of these cases by including a **TRY/FINALLY** block to clean up any spawned threads if a cancelation or other exception occurs. If unexpected exceptions do not concern you, then your code can perform two steps. They are disabling cancelability before threads are spawned, followed by enabling cancelability after the join operation finishes and after testing for any pending cancel operations. Following this disable and enable sequence prevents routine **pthread_join()** from producing a cancel point in a manager thread that has spawned threads that, in turn, share thread handles with the manager thread.

**Transitioning from Client to Server to Client**

Immediately before the program changes from a client to a server, it must obtain a handle on its environment as a client by calling **rpc_ss_get_thread_handle()**. When it reverts from a server to a client, it must reestablish the client environment by calling the **rpc_ss_set_thread_handle()** routine, supplying the previously obtained handle as a parameter.

**Syntax for Thread Routines**

The syntax for the **rpc_ss_get_thread_handle()** routine is:

```
rpc_ss_thread_handle_t  rpc_ss_get_thread_handle(void);
```

The syntax for the **rpc_ss_set_thread_handle()** routine is:

```
void rpc_ss_set_thread_handle (
    rpc_ss_thread_handle_t id
    );
```

The **rpc_ss_thread_handle_t** value identifies the thread to the RPC stub support library. The *id* parameter indicates the thread handle passed to the spawned thread by its creator, or the thread handle returned by the previous call to **rpc_ss_get_thread_handle()**.

**Rules for Using the Memory Management Routines:** You can use the **rpc_ss_allocate()** routine in the following:

- The manager code for an operation that has a full pointer in its argument list.
- The manager code for an operation to which the **enable_allocate** ACF attribute is applied.
- Code that is not called from a server stub but that has called the **rpc_ss_enable_allocate()** routine.
- A thread, spawned by code of any of the previous three types, that has made a call to the **rpc_ss_set_thread_handle()** routine using a thread handle obtained by this code.

**Examples Using Pointers:** The example application in this subsection contains the following files, listed here with the function of each file:

| Example | Function |
|---|---|
| **string_tree.idl** | Defines data types and interfaces (Figure 48 on page 262). |
| **string_tree_client.c** | The user of the interface (Figure 49 on page 263). |
| **string_tree_manager.c** | The server code that contains the procedure (Figure 50 on page 265). |
| **string_tree_server.c** | Declares the server; enables the client code to find the interface it needs (Figure 51 on page 266). |
| **string_tree output** | Shows the output (Figure 52 on page 267). |

**Note:** Source code for these examples can be found in the **/usr/lpp/dce/examples** directory.

```
[uuid(0144d600-2d28-11c9-a812-08002b0ecef1), version(0)]
interface string_tree
{
  /*
   * Maximum length of a string in the tree
   */
  const long int st_c_name_len = 32;

  /*
   * Definition of a node in the tree.
   */
  typedef struct node
  {
      [string] char name[0..st_c_name_len];
      [ptr] struct node *left;
      [ptr] struct node *right;
  } st_node_t;


  /*
   * Operation that prunes the left subtree of the specified
   * tree and returns it as the value.
   */
  st_node_t *st_prune_left (
      [in, out] st_node_t *tree /* root of tree by ref */
      );
}
```

*Figure 48. string_tree.idl Source*

```
#include <stdio.h>
#include <locale.h>
#include "sttree.h"

#include <stdlib.h>

/*
** Routine to print a depiction of the tree
*/
void st_print_tree (tree, indent)
  st_node_t *tree;
  int  indent;
{
  int i;
  if (tree == NULL) return;
  for (i = 0; i < indent; i++) printf("    ");
  printf("%s\n",tree->name);
  st_print_tree(tree->left, indent + 1);
  st_print_tree(tree->right, indent + 1);
}


/*
** Create a tree with a few nodes
*/
st_node_t *st_make_tree()
{
  st_node_t *root = (st_node_t *)malloc(sizeof(st_node_t));
  strcpy(root->name,"Root Node");

  /* left subtree node */
  root->left =  (st_node_t *)malloc(sizeof(st_node_t));
  strcpy(root->left->name,"Left subtree");

  /* left subtree children */
  root->left->right = NULL;
  root->left->left = (st_node_t *)malloc(sizeof(st_node_t));
  strcpy(root->left->left->name,"Child of left subtree");
  root->left->left->left = NULL;
  root->left->left->right = NULL;

  /* right subtree node */
  root->right =  (st_node_t *)malloc(sizeof(st_node_t));
  strcpy(root->right->name,"Right subtree");
  root->right->left = NULL;
  root->right->right = NULL;

  return root;
}
```

*Figure 49 (Part 1 of 2). string_tree_client.c Source*

```
int main(int argc, char *argv[])
{
  st_node_t *tree;
  st_node_t *subtree;

  setlocale(LC_ALL, "");

  /* setup and print original tree */
  tree = st_make_tree();
  printf("Original Tree:\n");
  st_print_tree(tree, 1);

  /* call the prune routine */
  subtree = st_prune_left (tree);

  /* print the resulting trees */
  printf("Pruned Tree:\n");
  st_print_tree(tree, 1);

  printf("Pruned subtree:\n");
  st_print_tree(subtree, 1);

  return(0);
  }
```

*Figure 49 (Part 2 of 2). string_tree_client.c Source*

```
#include <stdio.h>
#include "sttree.h"

/*
** Prune the left subtree of the specified tree and return
** it as the function value.
*/
st_node_t *st_prune_left (tree)
    st_node_t *tree;
{
  st_node_t *left_sub_tree = tree->left;
  tree->left = (st_node_t *)NULL;
  return left_sub_tree;
}
```

*Figure 50. string_tree_manager.c Source*

```
#include <stdio.h>
#include <locale.h>
#include "sttree.h"                  /* header created by idl compiler */
#define check_error(s, msg) if(s != rpc_s_ok) \
        {fprintf(stderr, "%s\n", msg); fflush(stderr); exit(1);}

int main(int argc, char **argv)
{
  unsigned32              status;         /* error status (nbase.h) */
  rpc_binding_vector_p_t  binding_vector; /* set of binding handles (rpc.h) */

  setlocale(LC_ALL, "");

  /* register interface with RPC runtime */
  rpc_server_register_if(string_tree_v0_0_s_ifspec, NULL, NULL, &status);
  check_error(status, "Can't register interface\n");

  /* establish protocol sequences */
  rpc_server_use_all_protseqs(rpc_c_protseq_max_calls_default,&status);
  check_error(status, "Can't establish protocol sequences\n");

  /* get set of this server's binding handles */
  rpc_server_inq_bindings(&binding_vector,&status);
  check_error(status, "Can't get binding handles\n");

  /* register addresses in endpoint map database */
  rpc_ep_register(string_tree_v0_0_s_ifspec,binding_vector,NULL,
    "string tree example",&status);
  check_error(status, "Can't add address to the endpoint database\n");

  /* establish namespace entry   */
  rpc_ns_binding_export(rpc_c_ns_syntax_dce,"/.:/string_tree",
   string_tree_v0_0_s_ifspec,binding_vector,NULL,
   &status);
  check_error(status, "Can't export to directory service\n");

  /* free set of binding handles */
  rpc_binding_vector_free(&binding_vector,&status);
  check_error(status, "Can't free binding handles and vector\n");

  /* listen for remote calls */
  rpc_server_listen(rpc_c_listen_max_calls_default,&status);
  check_error(status, "rpc listen failed\n");

return(0);
}
```

*Figure 51. string_tree_server.c Source*

```
Original Tree:
    Root Node
       Left subtree
           Child of left subtree
       Right subtree
Pruned Tree:
    Root Node
        Right subtree
Pruned subtree:
    Left subtree
        Child of left subtree
```

*Figure 52. string_tree output*

## Customized Handles

The **handle** attribute specifies that the type being declared is a user-defined, nonprimitive handle type, and is to be used in place of the predefined primitive handle type **handle_t**. The term *customized handle* denotes a nonprimitive handle.

The following example declares a customized handle type **filehandle_t**, a structure containing the textual representations of a host and a path name:

```
typedef [handle] struct {
     char host[256];
     char path[1024];
     } filehandle_t;
```

If the handle parameter is the first parameter in the list, then it is a customized handle that is used to determine the binding for the call. It must have the **in** attribute or the **in,out** attributes. A handle parameter that is not the first parameter in the parameter list need not have the **in** or **in,out** attributes.

Note that a **handle_t** parameter that is the first parameter in the list must not have the **transmit_as** attribute.

To build an application that uses customized handles, write custom binding and unbinding routines, and them with your application client code. At run time, each time the client calls an operation that uses a customized handle, the client stub calls the custom binding routine before it sends the RPC request. The client stub calls the custom unbinding routine after it receives a response.

The following paragraphs specify C prototypes for customized binding and unbinding routines; in these prototypes, *CUSTOM* is the name of the customized handle type.

The custom binding routine *CUSTOM*_**bind** generates a primitive binding handle from a customized handle and returns the primitive binding handle:

**handle_t** *CUSTOM*_**bind** (*CUSTOM c-handle* **)**

The custom unbinding routine *CUSTOM*_**unbind** takes two inputs, a customized handle and the primitive binding handle that was generated from it, and has no outputs:

**void** *CUSTOM*_**unbind (**
*CUSTOM c-handle* **,**
**handle_t** *rpc-handle***)**

A custom unbinding routine typically frees the primitive binding handle and any unneeded resources associated with the customized handle, but it is not required to do anything.

Because the **handle** attribute can occur only in a type declaration, a customized handle must have a named type. Because customized handle type names are used to construct custom binding and unbinding routine names, they cannot exceed 24 characters.

A customized handle can be coded either in a parameter list as an explicit handle or in an interface header as an implicit handle.

## Context Handles

Manager code often maintains state information for a client. A handle to this state information is passed to the client in an output parameter or as an operation result. The client passes the unchanged handle-to-the-state information as an input or input/output parameter of a subsequent manager operation that the client calls to manipulate that data structure. This handle-to-the-state information is called a *context handle*. A context handle is an untyped pointer or a pointer to a structure by tag name.

The manager causes the untyped pointer to point to the state information it will need the next time the client asks the manager to manipulate the context. For the client, the context handle is an opaque pointer (**idl_void_p_t** or an opaque structure tag). The client receives or supplies the context handle by means of the parameter list, but does not perform any transformations on it.

The RPC runtime maintains the context handle, providing an association between the client and the address space running the manager and the state information within that address space.

If a manager supports multiple interfaces, and a client obtains a context handle by performing an operation from one of these interfaces, the client can then supply the context handle to an operation from another of these interfaces.

A context handle can only be exchanged between the server process that created it and the client process for whom it was created. No other client except the one that obtained the context handle can use it without causing an application error.

### The context_handle Attribute:

Specify a context handle by either of the following methods:

- Use the **context_handle** attribute on a type that is defined as **void ***.
- Use the **context_handle** attribute on a parameter of type **void ***.
- Use the **context_handle** attribute on a type that is defined as a pointer to a structure by tag name.

For example, in the IDL file, you can define a context handle as follows:

- Within a type declaration:

  ```
  typedef [context_handle] void * my_context;
  ```

- Within a parameter declaration:

  ```
  [in, context_handle] void * my_context;
  ```

You can also define a context handle within a type declaration as a forward reference to a structure type by tag, as follows:

```
typedef [context_handle] struct opaque_struct * opaque_ch_t;
```

Note that you do not need to define the structure type in the IDL file; it is a forward reference to a structure whose definition can be included into the server code, either from a private **.h** file or from a server IDL file. As a result, the structure type is opaque to the client. This method of defining a context handle provides type checking and permits the server code to avoid extensive casting when manipulating the context handle.

A structure type in a context handle type definition must be referenced by tag name and not by type name. So, for example, the first of the following declarations is valid, while the second is not:

```
typedef [context_handle] struct struct_tag * valid_ch_t; / *valid */

typedef [context_handle] struct_type * invalid_ch_t; /* error */
```

The following example illustrates context handles defined as untyped pointers and as pointers to structures by tag name.

```
/* A context handle implemented as untyped pointer */
typedef [context_handle] void * void_ch_t;

/* A context handle implemented as a pointer to a structure by tag name */
typedef [context_handle] struct opaque_struct * opaque_ch_t;

/* Operations using both types of context handles */
void ch_oper(
     [in] void_ch_t v1,
     [in,out] void_ch_t *v2,
     [out] void_ch_t *v3,
     [in] opaque_ch_t *o2,
     [out] opaque_ch_t *o3
);

void_ch_t void_ch_oper ([in] handle_t h);

opaque_ch_t opaque_ch_oper([in] handle_t h);
```

It is possible to define a structure type in a context handle in the IDL file; for example, the following structure definition

```
typedef struct struct_tag {long l;} struct_type;
```

can either precede or follow the definition of **valid_ch_t** in the example previously shown. However, this practice is not recommended, since it violates the opaqueness of the context handle type.

The type name in a context handle declaration must be no longer than 23 characters.

The first operation on a context creates a context handle that the server procedure passes to the client. The client then passes the unmodified handle back to the server in a subsequent remote call. The called procedure interprets the context handle. For example, to specify a procedure that a client can use to obtain a context handle, you can define the following:

```
typedef [context_handle] void * my_context;
void op1(
   [in]handle_t h,
   [out] my_context * this_object);
```

To specify a procedure that a client can call to make use of a previously obtained context handle, you can define the following:

- Within a type declaration:

  ```
  void op2([in] my_context this_object);
  ```

- Within a parameter declaration:

  ```
  void op2([in, context_handle] void * this_object);
  ```

To close a context, and to clean the context on the client side, you can define the following:

- Within a type declaration:

  ```
  [in, out, my_context * this_object;
  ```

- Within a parameter declaration:

  ```
  void op3([in, out, context_handle] void ** this_object);
  ```

The resources associated with a context handle are reclaimed when, and only when, the manager changes the value of the **in,out** context handle parameter from non-**NULL** to **NULL**.

**The Context Rundown Procedure:** Some uses of context handles may require you to write a context rundown procedure in the application code for the server. If communications between the client and server are broken while the server is maintaining context for the client, RPC runs the context rundown procedure on the server to recover the resources represented by the context handle. If you declare a context handle as a named type, you must supply a rundown procedure for that type.

When a context requires a context rundown procedure, you must define a named type that has the **context_handle** attribute. For each different context handle type, you must provide a context rundown procedure as part of the manager code.

The format for the rundown procedure name is as follows:

*context_type_name_***rundown**

A rundown procedure takes one parameter, the handle of the context to be run down, and delivers no result. For example, if you declare the following:

```
typedef [context_handle] void * my_context;
```

the rundown procedure is as follows:

```
void my_context_rundown (my_context this_object);
```

Server application code that uses a certain context handle may be running in one or more server threads at the time that RPC detects that communications between the server and the client that is using that context have broken. The context rundown routine will not be invoked until a return of control to the server stub has happened in each of the threads that were using the context handle.

If application code in any of these threads destroys the context before returning control to the server stub from which it was called, your context rundown procedure will not be processed.

**Creating New Context:** When a client makes its first request to the manager to manipulate context, the manager creates context information and returns this information to the client through a parameter of the type **context_handle**. This parameter must be an output parameter or an input/output parameter whose value is **NULL** when the call is made. A context handle can also be a function result.

**Reclaiming Client Memory Resources for the Context Handle:** If a communications error causes the context handle to be unusable, the resources that maintain the context handle must be reclaimed. Use the **rpc_ss_destroy_client_context()** routine in the client application to reclaim the client side resources and to set the context handle value to **NULL**.

The syntax of the routine is as follows:

**void rpc_ss_destroy_client_context(**
    **void** **p_unusable_context_handle);*

**Relationship of Context Handles and Binding:** For the client, the context handle specifies the state within a server, and also contains binding information. If an operation has an input context handle or input/output context handle that is not **NULL**, no other binding information is required. A context handle that has only the **in** attribute cannot be **NULL**. If an operation has **in,out** context handle parameters but no **in** context handle parameters, at least one of the **in,out** context handle parameters cannot be **NULL**. However, if the only context handle parameters in an operation are written out, they carry no binding information. In this case, you must use another method to bind the client to a server.

If you specify multiple context handles in an operation, all active context handles must map to the same remote address space on the same server or the call fails. (A context handle is active while it represents context information that the server maintains for the client. It is inactive if no context has yet been created, or if the context is no longer in use.)

**Rules for Using Context Handles:** The following rules apply to using context handles:

- A context handle can be a parameter or a function result. You cannot use a context handle as an array element, as a structure or union member, or as the element type of a pipe.
- A context handle cannot have the **transmit_as** or **ptr** attributes.
- An input-only context handle cannot be **NULL**.
- A context handle cannot be pointed to, except by a top-level reference pointer.

**Examples Using Context Handles:** Figure 53 on page 272 shows a sample IDL file that uses context handles. Figure 54 on page 273 shows a sample context rundown procedure file.

```
/*
 * Filename: context_handle.idl
 */
[uuid(f38f5080-2d27-11c9-a96d-08002b0ecef1),
 pointer_default(ref), version (1.0)]
interface files
{
/* File context handle type */
typedef [context_handle] void * file_handle_t;
/* File specification type */
typedef [string] char * filespec_t;
/* File read buffer type */
typedef [string] char buf_t[*];

 /*
  * The file_open call requires that the client has located a
  * file server interface files and that an RPC handle that is
  * bound to that server be passed as the binding parameter h.
  *
  * Operation to OPEN a file; returns context handle for that file.
  */
file_handle_t file_open
(
 /* RPC handle bound to file server */
    [in] handle_t h,
 /* File specification of file to open */
    [in] filespec_t fs
);

 /*
  * The file_read call is able to use the context handle obtained
  * from the file_open as the binding parameter, thus an RPC
  * handle is not necessary.
  *
  * Operation to read from an opened file; returns true if not
  * end-of-file
  */
boolean file_read
(
 /* Context handle of opened file */
    [in] file_handle_t fh,
 /* Maximum number of characters to read */
    [in] long buf_size,
 /* Actual number of characters of data read */
    [out] long *data_size,
 /* Buffer for characters read */
    [out, size_is(buf_size), length_is(*data_size)] buf_t buffer
);
 /* Operation to close an opened file */
void file_close
(
 /* Valid file context handle goes [in]. On successful close,
  * null is returned.
  */
    [in,out] file_handle_t *fh
);
}
```

*Figure 53. Example of an IDL File That Uses a Context Handle*

```
/*
 * fh_rundown.c:  A context rundown procedure.
 */

#include <stdio.h>
#include "context_handle.h"     /* IDL-generated header file */

void file_handle_t_rundown
(
    file_handle_t file_handle   /* Active context handle
                                 * (open file handle) */
)

{
    /*
     * This procedure is called by the RPC runtime on the SERVER
     * side when communication is broken between the client and
     * server. This gives the server the opportunity to reclaim
     * resources identified by the passed context handle.  In
     * this case, the passed context handle identifies a file,
     * and simply closing the file cleans up the state maintained
     * by the context handle, that is "runs down" the context handle.
     * Note that the file_close manager operation is not used here;
     * perhaps it could be, but it is more efficient to use the
     * underlying file system call to do the close.
     *
     * File handle is void*, it must be cast to FILE*
     */
    fclose((FILE *)file_handle);
}
```

*Figure 54. Example of a Context Rundown Procedure*

## Associating a Data Type with a Transmitted Type

The **transmit_as** attribute associates a *transmitted type* that stubs pass over the network with a *presented type* that clients and servers manipulate. The specified transmitted type must be a named type defined previously in another type declaration.

There are two primary uses for this attribute:

- To pass complex data types for which the IDL compiler cannot generate marshalling and unmarshalling code.
- To pass data more efficiently. An application can provide routines to convert a data type between a sparse representation (presented to the client and server programs) and a compact one (transmitted over the network).

To build an application that uses presented and transmitted types, you must write routines to perform conversions between the types and to manage storage for the types, and you must link those routines with your application code. At run time, the client and server stubs call these routines before sending and after receiving data of these types.

The following paragraphs specify C prototypes for generic binding and unbinding routines. In these prototypes, *PRES* is the name of the presented type, and *TRANS* is the name of the transmitted type.

The *PRES*_**to_xmit** routine allocates storage for the transmitted type and converts from the presented type to the transmitted type:

**void** *PRES*_**to_xmit** (*PRES *presented*, *TRANS* *******transmitted***)**

The *PRES*_**from_xmit** routine converts from the transmitted type to the presented type and allocates any storage referred by pointers in the presented type:

**void** *PRES*_**from_xmit** (*TRANS* ******transmitted*, *PRES* ******presented***)**

The *PRES*_**free_inst** routine frees any storage referred by pointers in the presented type by *PRES*_**from_xmit**:

**void** *PRES*_**free_inst** (*PRES* ******presented***)**

Suppose that the **transmit_as** attribute appears either on the type of a parameter, or on a component of a parameter, and that the parameter has the **out** or **in,out** attribute. Then the *PRES*_**free_inst** routine will be called automatically for the data item that has the **transmit_as** attribute.

Suppose the **transmit_as** attribute appears on the type of a parameter, and that the parameter has only the **in** attribute. Then the *PRES*_**free_inst** routine will be called automatically.

Finally, suppose that the **transmit_as** attribute appears on a component of a parameter, and that the parameter only has the **in** attribute. Then the *PRES*_**free_inst** routine will not be called automatically for the component. The manager application code must release any resources that the component uses, possibly by explicitly calling the *PRES*_**free_inst** routine.

The *PRES*_**free_xmit** routine frees any storage that has been allocated for the transmitted type by *PRES*_**to_xmit**:

**void** *PRES*_**free_xmit** (*TRANS* ******transmitted***)**

A type with the **transmit_as** attribute cannot have other type attributes, specifically:

- A pipe type.
- A pipe element type.
- A type with the **context_handle** attribute.
- A type of which any instance has the **context_handle** attribute.
- A type that includes the **[handle]** attribute in its definition cannot be used, directly or indirectly, in the definition of a type with the **[transmit_as]** attribute. Nor can a type that includes the **[transmit_as]** attribute in its definition be used, directly or indirectly, in the definition of a type with the **[handle]** attribute.
- A conformant array type.
- A varying array type.
- A structure type containing a conformant array.
- An array type of which any instance is varying.
- A type with the **represent_as** attribute.

  The type name in a declaration for a **transmit_as** attribute cannot exceed 21 characters.

A transmitted type specified by the **transmit_as** attribute must be either a base type, a predefined type, or a named type defined using **typedef**. A transmitted type cannot be a conformant array type or a conformant structure type if any instance of that type is an **in** parameter or an **in, out** parameter.

The following is an example of **transmit_as**.

Assuming the following declarations:

```
typedef
   struct tree_node_t {
      data_t data;
      struct tree_node_t * left;
      struct tree_node_t * right;
   } tree_node_t;

typedef
   [transmit_as(tree_xmit_t)] tree_node_t *tree_t;
```

The application code must include routines that match the prototypes:

```
void tree_t_to_xmit ( tree_t *, (tree_xmit_t **) );
void tree_t_from_xmit ( (tree_xmit_t *), (tree_t *) );
void tree_t_free_inst ( tree_t *);
void tree_t_free_xmit ( (tree_xmit_t *) );
```

## IDL Grammar Synopsis

This section summarizes the IDL syntax in extended Backus-Naur Format (BNF) notation.

*Table 11 (Page 1 of 8). Backus-Naur Format for the Interface Definition Language*

| Number | Production Rule | | |
|---|---|---|---|
| 1 | *<interface>* | ::= | *<interface_init>* *<interface_start>* *<interface_tail>* |
| 2 | *<interface_start>* | ::= | *<interface_attributes>* **interface IDENTIFIER** |
| 3 | *<interface_init>* | ::= | ϕ |
| 4 | *<interface_tail>* | ::= | **{** *<interface_body>* **}**<br>\| error<br>\| error **}** |
| 5 | *<interface_body>* | ::= | *<optional_imports>* *<exports>* *<extraneous_semi>* |
| 6 | *<optional_imports>* | ::= | *<imports>*<br>\| ϕ |
| 7 | *<imports>* | ::= | *<import>*<br>\| *<imports>* *<import>* |
| 8 | *<import>* | ::= | **import** error<br>\| **import** error **;**<br>\| **import** *<import_files>* **;** |
| 9 | *<import_files>* | ::= | *<import_file>*<br>\| *<import_files>* **,** *<import_file>* |
| 10 | *<import_file>* | ::= | **STRING** |
| 11 | *<exports>* | ::= | *<export>*<br>\| *<exports>* *<extraneous_semi>* *<export>* |

*Table 11 (Page 2 of 8). Backus-Naur Format for the Interface Definition Language*

| **Number** | **Production Rule** | | |
|---|---|---|---|
| 12 | *<export>* | ::= | *<type_dcl>* **;**<br>\| *<const_dcl>* **;**<br>\| *<operation_dcl>* **;**<br>\| error **;** |
| 13 | *<const_dcl>* | ::= | **const** *<type_spec>* *<declarator>* **=** *<const_exp>* |
| 14 | *<const_exp>* | ::= | *<expression>* |
| 15 | *<type_dcl>* | ::= | **typedef** *<type_declarator>* |
| 16 | *<type_declarator>* | ::= | *<attributes>* *<type_spec>* *<declarators>* |
| 17 | *<type_spec>* | ::= | *<simple_type_spec>*<br>\| *<constructed_type_spec>* |
| 18 | *<simple_type_spec>* | ::= | *<floating_point_type_spec>*<br>\| *<integer_type_spec>*<br>\| *<char_type_spec>*<br>\| *<boolean_type_spec>*<br>\| *<byte_type_spec>*<br>\| *<void_typ/_spec>*<br>\| *<named_type_spec>*<br>\| *<handle_type_spec>* |
| 19 | *<constructed_type_spec>* | ::= | *<struct_type_spec>*<br>\| *<union_type_spec>*<br>\| *<enum_type_spec>*<br>\| *<pipe_type_spec>* |
| 20 | *<named_type_spec>* | ::= | **IDENTIFIER** |
| 21 | *<floating_point_type_spec>* | ::= | **float**<br>\| **double** |
| 22 | *<extraneous_comma>* | ::= | $\phi$<br>\| **,** |
| 23 | *<extraneous_semi>* | ::= | $\phi$<br>\| **;** |
| 24 | *<optional_unsigned_kw>* | ::= | **unsigned**<br>\| $\phi$ |
| 25 | *<integer_size_spec>* | ::= | **small**<br>\| **short**<br>\| **long**<br>\| **hyper** |
| 26 | *<integer_modifiers>* | ::= | *<integer_size_spec>*<br>\| **unsigned** *<integer_size_spec>*<br>\| *<integer_size_spec>* **unsigned** |
| 27 | *<integer_type_spec>* | ::= | *<integer_modifiers>*<br>\| *<integer_modifiers>* **int**<br>\| *<optional_unsigned_kw>* **int** |

*Table 11 (Page 3 of 8). Backus-Naur Format for the Interface Definition Language*

| Number | Production Rule | | |
|---|---|---|---|
| 28 | *<char_type_spec>* | ::= | *<optional_unsigned_kw>* **char** |
| 29 | *<boolean_type_spec>* | ::= | **boolean** |
| 30 | *<byte_type_spec>* | ::= | **byte** |
| 31 | *<void_type_spec>* | ::= | **void** |
| 32 | *<handle_type_spec>* | ::= | **handle_t** |
| 33 | *<push_name_space>* | ::= | **{** |
| 34 | *<pop_name_space>* | ::= | **}** |
| 35 | *<union_type_spec>* | ::= | **union** *<ne_union_body>* <br>\| **union switch (** *<disc_type_spec>* **IDENTIFIER )** *<union_body>* <br>\| **union IDENTIFIER** *<ne_union_body>* <br>\| **union switch (** *<disc_type_spec>* <br>\| **union IDENTIFIER switch (** *<disc_type_spec>* **IDENTIFIER )** *<union_body>* <br>**union IDENTIFIER** |
| 36 | *<disc_type_spec>* | ::= | *<simple_type_spec>* |
| 37 | *<ne_union_body>* | ::= | *<push_name_space>* *<ne_union_cases>* *<pop_name_space>* |
| 38 | *<union_body>* | ::= | *<push_name_space>* *<union_cases>* *<pop_name_space>* |
| 39 | *<ne_union_cases>* | ::= | *<ne_union_case>* <br>\| ne_union_cases extraneous_semi ne_union case |
| 40 | *<union_cases>* | ::= | *<union_case>* <br>\| *<union_cases>* *<extraneous_semi>* *<union_case>* |
| 41 | *<ne_union_case>* | ::= | *<ne_union_member>* |
| 42 | *<union_case>* | ::= | *<union_case_list>* *<union_member>* |
| 43 | *<ne_union_case_list>* | ::= | *<ne_union_case_label>* <br>\| *<ne_union_case_list>* **,** ne_union_case_label |
| 44 | *<union_case_list>* | ::= | *<union_case_label>* <br>\| *<union_case_list>* *<union_case_label>* |
| 45 | *<ne_union_case_label>* | ::= | *<const_exp>* |
| 46 | *<union_case_label>* | ::= | **case** *<const_exp>* **:** <br>\| **default :** |
| 47 | *<ne_union_member>* | ::= <br>::= | *<attribute_opener>* *<rest_of_attribute_list>* <br>*<attribute_opener>* *<rest_of_attribute_list>* *<type_spec>* *<declarator>* **;** |
| 48 | *<union_member>* | ::= | **;** <br>\| *<attributes>* *<type_spec>* *<declarator>* **;** |

*Table 11 (Page 4 of 8). Backus-Naur Format for the Interface Definition Language*

| Number | Production Rule | | |
|---|---|---|---|
| 49 | *<struct_type_spec>* | ::= | **struct** *<push_name_space> <member_list> <pop_name_space>* |
| | | | \| **struct IDENTIFIER** *<push_name_space> <member_list> <pop_name_space>* |
| | | | \| **struct IDENTIFIER** |
| 50 | *<member_list>* | ::= | *<member>* |
| | | | \| *<member_list> <extraneous_semi> <member>* |
| 51 | *<member>* | ::= | *<attributes> <type_spec> <old_attribute_syntax> <declarators>* **;** |
| 52 | *<enum_type_spec>* | ::= | **enum** *<enum_body>* |
| 53 | *<enum_body>* | ::= | **{** *<enum_ids>* **}** |
| 54 | *<enum_ids>* | ::= | ϕ |
| | | | \| *<enum_id>* |
| | | | \| *<enum_ids>* **,** *<extraneous_comma> <enum_id>* |
| 55 | *<enum_id>* | ::= | **IDENTIFIER** |
| 56 | *<pipe_type_spec>* | ::= | **pipe** *<type_spec>* |
| 57 | *<declarators>* | ::= | *<declarator>* |
| | | | \| *<declarators>* **,** *<extraneous_comma> <declarator>* |
| 58 | *<declarator>* | ::= | *<direct_declarator>* |
| | | | \| *<pointer> <direct_declarator>* |
| 59 | *<pointer>* | ::= | ***** |
| | | | \| ***** *<pointer>* |
| 60 | *<direct_declarator>* | ::= | **IDENTIFIER** |
| | | | \| *<direct_declarator> <array_bounds>* |
| | | | \| **(** *<declarator>* **)** |
| | | | \| *<direct_declarator> <parameter_dcls>* |
| 61 | *<array_bounds>* | ::= | **[ ]** |
| | | | \| **[ * ]** |
| | | | \| **[** *<const_exp>* **]** |
| | | | \| **[ * .. * ]** |
| | | | \| **[ * ..** *<const_exp>* **]** |
| | | | \| **[** *<const_exp>* **.. * ]** |
| | | | \| **[** *<const_exp>* **..** *<const_exp>* **]** |
| 62 | *<operation_dcl>* | ::= | *<attributes> <type_spec> <declarators>* |
| | | | \| error *<declarators>* |
| 63 | *<parameter_dcls>* | ::= | *<param_names> <param_list> <end_param_names>* |
| 64 | *<param_names>* | ::= | **(** |
| 65 | *<end_param_names>* | ::= | *<extraneous_comma>* **)** |
| 66 | *<param_list>* | ::= | *<param_dcl>* |
| | | | \| *<param_list>* **,** *<extraneous_comma> <param_dcl>* |

Table 11 (Page 5 of 8). Backus-Naur Format for the Interface Definition Language

| Number | Production Rule | | |
|---|---|---|---|
| | | | \| φ |
| 67 | *<param_dcl>* | ::= | *<attributes> <type_spec> <old_attribute_syntax> <declarator_or_null>*<br>\| error *<old_attribute_syntax> <declarator_or_null>* |
| 68 | *<declarator_or_null>* | ::= | *<declarator>*<br>\| φ |
| 69 | *<attribute_opener>* | ::= | **[** |
| 70 | *<attribute_closer>* | ::= | **]** |
| 71 | *<bounds_opener>* | ::= | **(** |
| 72 | *<bounds_closer>* | ::= | **)** |
| 73 | *<old_attribute_syntax>* | ::= | *<attributes>* |
| 74 | *<interface_attributes>* | ::= | *<attribute_opener> <interface_attr_list> <attribute_closer>*<br>\| *<attribute_opener>* error *<attribute_closer>*<br>\| φ |
| 75 | *<interface_attr_list>* | ::= | *<interface_attr>*<br>\| *<interface_attr_list>* **,** *<extraneous_comma> <interface_attr>*<br>\| φ |
| 76 | *<interface_attr>* | ::= | **uuid** error<br>\| **uuid UUID_REP**<br>\| **endpoint (** *<port_list>* **)**<br>\| **exceptions (** *<excep_list>* **)**<br>\| **version (** *<version_number>* **)**<br>\| **local**<br>\| **pointer_default (** *<pointer_class>* **)** |
| 77 | *<pointer_class>* | ::= | **ref**<br>\| **ptr**<br>\| **unique** |
| 78 | *<version_number>* | ::= | **INTEGER**<br>\| **FLOAT** |
| 79 | *<port_list>* | ::= | *<port_spec>*<br>\| *<port_list>* **,** *<extraneous_comma> <port_spec>* |
| 80 | *<excep_list>* | ::= | *<excep_spec>*<br>\| *<excep_list>* **,** *<extraneous_comma> <excep_spec>* |
| 81 | *<port_spec>* | ::= | **STRING** |
| 82 | *<excep_spec>* | ::= | **IDENTIFIER** |
| 83 | *<fp_attribute>* | ::= | *<array_bound_type> <bounds_opener> <array_bound_id_list> <bounds_closer>* |
| 84 | *<array_bound_type>* | ::= | **first_is** |

Table 11 (Page 6 of 8). Backus-Naur Format for the Interface Definition Language

| Number | Production Rule | | |
|---|---|---|---|
| | | | \| **last_is**<br>\| **length_is**<br>\| **max_is**<br>\| **min_is**<br>\| **size_is** |
| 85 | *<array_bound_id_list>* | ::= | *<array_bound_id>*<br>\| *<array_bound_id_list>* **,** *<array_bound_id>* |
| 86 | *<array_bound_id>* | ::= | **IDENTIFIER**<br>\| *** IDENTIFIER**<br>\| $\phi$ |
| 87 | *<neu_switch_type>* | ::= | **switch_is** |
| 88 | *<neu_switch_id>* | ::=<br>: | **IDENTIFIER**<br>\| ***IDENTIFIER** |
| 89 | *<attributes>* | ::= | *<attribute_opener>* *<rest_of_attribute_list>*<br>\| $\phi$ |
| 90 | *<rest_of_attribute_list>* | ::= | *<attribute_list>* *<attribute_closer>*<br>\| error *<attribute_closer>*<br>\| error **;** |
| 91 | *<attribute_list>* | ::= | *<attribute>*<br>\| *<attribute_list>* **,** *<extraneous_comma>* *<attribute>* |
| 92 | *<attribute>* | ::= | *<fp_attribute>*<br>\| **broadcast**<br>\| **maybe**<br>\| **idempotent**<br>\| **reflect_deletions**<br>\| **ptr**<br>\| **in**<br>\| **in ( shape )**<br>\| **out**<br>\| **out ( shape )**<br>\| **v1_array**<br>\| **string**<br>\| **v1_string**<br>\| **unique**<br>\| **ref**<br>\| **ignore**<br>\| **context_handle**<br>\| **v1_struct**<br>\| **v1_enum**<br>\| **align ( small )**<br>\| **align ( short )**<br>\| **align ( long )**<br>\| **align ( hyper )**<br>\| **handle**<br>\| **transmit_as (** *<simple_type_spec>* **)**<br>\| **switch_type (** *<simple_type_spec>***)**<br>\| **case (** *<ne_union_case_list>* **)**<br>\| **default** |

*Table 11 (Page 7 of 8). Backus-Naur Format for the Interface Definition Language*

| **Number** | **Production Rule** | | |
|---|---|---|---|
| | | | \| **IDENTIFIER** |
| 93 | *<expression>* | ::= | *<conditional_expression>* |
| 94 | *<conditional_expression>* | ::= | *<logical_OR_expression>*<br>\| *<logical_OR_expression>* **?** *<expression>* **:** *<conditional_expression>* |
| 95 | *<logical_OR_expression>* | ::= | *<logical_AND_expression>*<br>\| *<logical_OR_expression>* **\|\|** *<logical_AND_expression>* |
| 96 | *<logical_AND_expression>* | ::= | *<inclusive_OR_expression>*<br>\| *<logical_AND_expression>* **&&** *<inclusive_OR_expression>* |
| 97 | *<inclusive_OR_expression>* | ::= | *<exclusive_OR_expression>*<br>\| *<inclusive_OR_expression>* **\|** *<exclusive_OR_expression>* |
| 98 | *<exclusive_OR_expression>* | ::= | *<AND_expression>*<br>\| *<exclusive_OR_expression>* ^ *<AND_expression>* |
| 99 | *<AND_expression>* | ::= | *<equality_expression>*<br>\| *<AND_expression>* **&** *<equality_expression>* |
| 100 | *<equality_expression>* | ::= | *<relational_expression>*<br>\| *<equality_expression>* **==** *<relational_expression>*<br>\| *<equality_expression>* **!=** *<relational_expression>* |
| 101 | *<relational_expression>* | ::= | *<shift_expression>*<br>\| *<relational_expression>* **<** *<shift_expression>*<br>\| *<relational_expression>* > *<shift_expression>*<br>\| *<relational_expression>* **<=** *<shift_expression>*<br>\| *<relational_expression>* >= *<shift_expression>* |
| 102 | *<shift_expression>* | ::= | *<additive_expression>*<br>\| *<shift_expression>* **<<** *<additive_expression>*<br>\| *<shift_expression>* >> *<additive_expression>* |
| 103 | *<additive_expression>* | ::= | *<multiplicative_expression>*<br>\| *<additive_expression>* **+** *<multiplicative_expression>*<br>\| *<additive_expression>* **-** *<multiplicative_expression>* |
| 104 | *<multiplicative_expression>* | ::= | *<cast_expression>*<br>\| *<multiplicative_expression>* ***** *<cast_expression>*<br>\| *<multiplicative_expression>* **/** *<cast_expression>*<br>\| *<multiplicative_expression>* **%** *<cast_expression>* |
| 105 | *<cast_expression>* | ::= | *<unary_expression>* |
| 106 | *<unary_expression>* | ::= | *<primary_expression>*<br>\| **+** *<primary_expression>*<br>\| **-** *<primary_expression>*<br>\| ˜ *<primary_expression>*<br>\| **!** *<primary_expression>* |
| 107 | *<primary_expression>* | ::= | **(** *<expression>* **)**<br>\| **INTEGER** |

*Table 11 (Page 8 of 8). Backus-Naur Format for the Interface Definition Language*

| Number | Production Rule |
|---|---|
| | \| **CHAR** |
| | \| **IDENTIFIER** |
| | \| **STRING** |
| | \| **NULL** |
| | \| **TRUE** |
| | \| **FALSE** |
| | \| **FLOAT** |

Table 12 is an index of the BNF production rules used in Table 11 on page 275. Use Table 12 as an index to find where a production rule is defined and used in Table 11 on page 275.

*Table 12 (Page 1 of 3). Cross Reference of the Backus-Naur Format for the Interface Definition Language*

| Production Rule | Defined | Used |
|---|---|---|
| *<additive_expression>* | 103 | *102, 103* |
| *<AND_expression>* | 99 | *98, 99* |
| *<array_bound_id>* | 86 | *85* |
| *<array_bound_id_list>* | 85 | *83, 85* |
| *<array_bound_type>* | 84 | *83* |
| *<array_bounds>* | 61 | *60* |
| *<attribute>* | 92 | *91* |
| *<attribute_closer>* | 70 | *74, 90* |
| *<attribute_list>* | 91 | *90, 91* |
| *<attribute_opener>* | 69 | *74, 89* |
| *<attributes>* | 89 | *16, 48, 51, 62, 67, 73* |
| *<boolean_type_spec>* | 29 | *18* |
| *<bounds_closer>* | 72 | *83* |
| *<bounds_opener>* | 71 | *83* |
| *<byte_type_spec>* | 30 | *18* |
| *<cast_expression>* | 105 | *104* |
| *<char_type_spec>* | 28 | *18* |
| *<conditional_expression>* | 94 | *93, 94* |
| *<const_dcl>* | 13 | *12* |
| *<const_exp>* | 14 | *13, 46, 61* |
| *<constructed_type_spec>* | 19 | *17* |
| *<declarator>* | 58 | *13, 48, 57, 60, 68* |
| *<declarator_or_null>* | 68 | *67* |
| *<declarators>* | 57 | *16, 51, 57, 62* |
| *<direct_declarator>* | 60 | *58, 60* |
| *<disc_type_spec>* | 36 | *35* |
| *<end_param_names>* | 65 | *63* |
| *<enum_body>* | 53 | *52* |
| *<enum_id>* | 55 | *54* |
| *<enum_ids>* | 54 | *53, 54* |
| *<enum_type_spec>* | 52 | *19* |
| *<equality_expression>* | 100 | *99, 100* |

*Table 12 (Page 2 of 3). Cross Reference of the Backus-Naur Format for the Interface Definition Language*

| Production Rule | Defined | Used |
|---|---|---|
| *<excep_list>* | 80 | *76* |
| *<excep_spec>* | 82 | *80* |
| *<exclusive_OR_expression>* | 98 | *97, 98* |
| *<export>* | 12 | *11* |
| *<exports>* | 11 | *5, 11* |
| *<expression>* | 93 | *14, 94, 107* |
| *<extraneous_comma>* | 22 | *54, 57, 65, 66, 75, 79, 91* |
| *<extraneous_semi>* | 23 | *5, 11, 40, 50* |
| *<floating_point_type_spec>* | 21 | *18* |
| *<fp_attribute>* | 83 | *92* |
| *<handle_type_spec>* | 32 | *18* |
| *<import>* | 8 | *7* |
| *<import_file>* | 10 | *9* |
| *<import_files>* | 9 | *8, 9* |
| *<imports>* | 7 | *6, 7* |
| *<inclusive_OR_expression>* | 97 | *96, 97* |
| *<integer_modifiers>* | 26 | *27* |
| *<integer_size_spec>* | 25 | *26* |
| *<integer_type_spec>* | 27 | *18* |
| *<interface>* | 1 | |
| *<interface_attr>* | 76 | *75* |
| *<interface_attr_list>* | 75 | *74, 75* |
| *<interface_attributes>* | 74 | *2* |
| *<interface_body>* | 5 | *4* |
| *<interface_init>* | 3 | *1* |
| *<interface_start>* | 2 | *1* |
| *<interface_tail>* | 4 | *1* |
| *<logical_AND_expression>* | 96 | *95, 96* |
| *<logical_OR_expression>* | 95 | *94, 95* |
| *<member>* | 51 | *50* |
| *<member_list>* | 50 | *49, 50* |
| *<multiplicative_expression>* | 104 | *103, 104* |
| *<named_type_spec>* | 20 | *18* |
| *<ne_union_body>* | 37 | *35* |
| *<ne_union_case>* | 41 | *37, 39* |
| *<ne_union_cases>* | 39 | *37* |
| *<ne_union_case_label>* | 45 | *43, 44* |
| *<ne_union_case_list>* | 43 | *44* |
| *<ne_union_member>* | 47 | *41* |
| *<neu_switch_id>* | 88 | |
| *<neu_switch_type>* | 87 | |
| *<old_attribute_syntax>* | 73 | *51, 67* |
| *<operation_dcl>* | 62 | *12* |

*Table 12 (Page 3 of 3). Cross Reference of the Backus-Naur Format for the Interface Definition Language*

| **Production Rule** | **Defined** | **Used** |
|---|---|---|
| *<optional_imports>* | 6 | *5* |
| *<optional_unsigned_kw>* | 24 | *27, 28* |
| *<param_dcl>* | 67 | *66* |
| *<param_list>* | 66 | *63, 66* |
| *<param_names>* | 64 | *63* |
| *<parameter_dcls>* | 63 | *60* |
| *<pipe_type_spec>* | 56 | *19* |
| *<pointer>* | 59 | *58, 59* |
| *<pointer_class>* | 77 | *76* |
| *<pop_name_space>* | 34 | *38, 49* |
| *<port_list>* | 79 | *76, 79* |
| *<port_spec>* | 81 | *79* |
| *<primary_expression>* | 107 | *106* |
| *<push_name_space>* | 33 | *38, 49* |
| *<relational_expression>* | 101 | *100, 101* |
| *<rest_of_attribute_list>* | 90 | *89* |
| *<shift_expression>* | 102 | *101, 102* |
| *<simple_type_spec>* | 18 | *17, 35, 92* |
| *<struct_type_spec>* | 49 | *19* |
| *<type_dcl>* | 15 | *12* |
| *<type_declarator>* | 16 | *15* |
| *<type_spec>* | 17 | *13, 16, 48, 51, 56, 62, 67* |
| *<unary_expression>* | 106 | *105* |
| *<union_body>* | 38 | *35* |
| *<union_case>* | 42 | *40* |
| *<union_case_label>* | 46 | *44* |
| *<union_case_list>* | 44 | *42, 44* |
| *<union_cases>* | 40 | *38, 40* |
| *<union_member>* | 48 | *42* |
| *<union_type_spec>* | 35 | *19* |
| *<version_number>* | 78 | *76* |
| *<void_type_spec>* | 31 | *18* |

# Chapter 12. Attribute Configuration Language

The Attribute Configuration Language is used for writing an attribute configuration file (ACF). Use the attributes in the ACF to change the interaction between the application code and stubs without affecting the client/server network interaction.

## Syntax Notation Conventions

The syntax of the Attribute Configuration Language is similar to the syntax of the Interface Definition Language (IDL). For syntax information, see “Syntax Notation Conventions” on page 221.

**Use of Brackets**

The use of **[ ]** (brackets) can be either a required part of the syntax or can denote that a string is optional to the syntax. To differentiate between the two, brackets that are required are shown as [ ] (regular type brackets). Brackets that contain optional strings are shown as *[ ]* (italicized brackets).

**Use of the Vertical Bar**

A **|** (vertical bar) denotes a logical OR.

## Attribute Configuration File (ACF)

The ACF changes the way the IDL compiler interprets the interface definition, written in the Interface Definition Language (IDL). The IDL file defines a means of interaction between a client and a server. For new server implementations to be compatible across the network with existing servers, the interaction between the client and server must not be changed. If the interaction between an application and a specific stub needs to change, you must provide an ACF when you build this stub.

The ACF affects only the interaction between the generated stub code and the local application code; it has no effect on the interaction between local and remote stubs. Therefore, client and server writers are likely to have different ACFs that they can change as desired.

## Naming the ACF

To name the ACF, if it is an HFS file, replace the extension of the IDL file (**.idl**) with the extension of the ACF (**.acf**). The ACF associated with *your_idl_filename***.idl** is *your_idl_filename***.acf**.

If it is a partitioned data set, replace the member name of the IDL data set with the same member name in the ACF data set. The IDL compiler searches the data set referred to by the DDNAME ACF with the same member name as in the IDL data set. For example, for an IDL file in USERPRFX.APPLNAME.IDL(BINOP), the IDL compiler searches for the ACF in USERPRFX.APPLNAME.ACF(BINOP).

## Compiling the ACF

When you issue the **idl** command, naming the IDL file to compile, the compiler searches for a corresponding ACF and compiles it along with the IDL file. the compiler also searches for any ACF (there can be more than one) associated with any imported IDL files. The stubs that the compiler creates contain the appropriate changes.

## ACF Features

The ACF attributes and the features associated with the attributes are as follows:

| | |
|---|---|
| **include** statement | Includes header files in the generated code |
| **auto_handle**, **explicit_handle**, **implicit_handle**, **binding_callout** | Controls binding |
| **comm_status**, **fault_status** | Indicates parameters to hold status conditions occurring in the call |
| **cs_char**, **cs_tag_rtn**, **cs_stag**, **cs_drtag**, **cs_rtag** | Controls the transmission of international (non-PCS) characters |
| **code**, **nocode** | Controls which operations of the IDL file are compiled |
| **encode**, **decode** | Controls the generation of IDL encoding services stubs to perform encoding or decoding operations |
| **extern_exceptions** | Indicates user-defined parameters to hold status conditions occurring in the call |
| **represent_as** | Controls conversion between local and network data types |
| **enable_allocate** | Forces the initialization of the memory management routines |
| **heap** | Specifies objects to be allocated from heap memory. **Note:** For performance reasons, use of the **heap** attribute on the z/OS DCE product is not recommended. |

## Structure

The structure of the ACF is:

*interface_header*
**{**
*interface_body*
**}**

Follow these structural rules when writing an attribute configuration file:

- The base name of the ACF must be the same as the base name of the IDL file although the extensions are different.
- The interface name in the ACF must be the same as the interface name in the corresponding IDL file.
- With a few exceptions, any type, parameter, or operation names in the ACF must be declared in the IDL file, or defined in files included by use of the **include** statement, as the same class of name.

- Except for additional status parameters, any parameter name that occurs within an operation in the ACF must also occur within that operation in the IDL file.

## ACF Interface Header

The ACF interface header has the following structure:

*[* **[** *acf_attribute_list* **]** *]* **interface** *idl_interface_name*

The *acf_attribute_list* is optional. The interface header attributes can include one or more of the following attributes, entered within brackets. If you use more than one attribute, separate them with commas and include the list within a single pair of brackets. (Note that some of these attributes can also be used in the ACF body as described in "ACF Interface Body.")

- **code**
- **nocode**
- **implicit_handle(** *handle_type handle_name* **)**
- **auto_handle**
- **explicit_handle**
- **encode**
- **decode**
- **binding_callout(** *routine_name* **)**
- **extern_exceptions(** *exception_name[*,*exception_name]...* **)**
- **cs_tag_rtn(** *tag_set_routine* **)**

The following example shows how to use more than one attribute in the ACF interface header:

```
[auto_handle, binding_callout(rpc_ss_bind_authn_client)] interface phone_direct
{
}
```

## ACF Interface Body

The ACF interface body can contain the elements in the following list. Note that some of the attributes listed here can also be used in the ACF header, as described in "ACF Interface Header." If you use more than one attribute, separate them with commas and include the list within a single pair of brackets.

- An **include** statement:

  **include "***filename***"** *[*, **"***filename***"** *]* . . .**;**

  **Note:** Omit the extension of the file name in an **include** statement; the IDL compiler appends the correct extension for the language you are using. The IDL compiler appends the **.h** extension.

- A declared type:

  **typedef** *[* **[represent_as (***local_type_name***) ]** | **[heap]** |
  **[cs_char (***local_type_name***)]***] type_name***;**

- An operation:

  *[* **[explicit_handle]** | **[comm_status]** | **[fault_status]** |
  **[code]** | **[nocode]** | **[enable_allocate]** |
  **[encode]** | **[decode]** | **[cs_tag_rtn (** *tag_set_routine***)]** *] operation_name* **(***[parameter_list]***);**

  A *parameter_list* is a list of zero or more parameter names as they appear in the corresponding operation definition of the IDL file. You do not need to use all the parameter names that occur in the IDL operation definition; use only those to which you attach an ACF attribute. If you use more than one parameter name, the names must be separated by commas.

- A parameter within an operation:

*[* **[comm_status]** | **[fault_status]** | **[heap]** | **[cs_stag]** | **[cs_drtag]** | **[cs_rtag]** *] parameter_name*

## The include Statement

This statement specifies any additional header files you want included in the generated stub code. You can specify more than one header file.

Use the **include** statement whenever you use the **represent_as**, **implicit_handle**, or **cs_char** attributes and the specified type is not defined or imported in the IDL file.

The **include** statement has the following syntax. (An example is shown with the **represent_as** example in "The represent_as Attribute" on page 293.)

**include "***filename***" ;**

## The auto_handle Attribute

This attribute causes the client stub and RPC runtime to manage the binding to the server by using a directory service. Any operation in the interface that has no parameter containing binding information is bound automatically to a server so the client does not have to specify a binding to a server.

When an operation is automatically bound, the client does not have to specify the server on where an operation runs. If you make a call on an operation without explicit binding information in an interface for which you have specified **auto_handle**, and no client/server binding currently exists, the RPC runtime system selects an available server and establishes a binding. This binding is used for this call and subsequent calls to all operations in the interface that do not include explicit binding information while the server is still available.

An abrupt server ending, network failure, or other problems can cause a break in binding. If this occurs during an automatically bound operation, RPC issues the call to another server, provided one is available, and the operation is idempotent or the RPC runtime system determines that the call did not start to run on the server. Similarly, if a communications or server failure occurs between calls, RPC binds to another server for the next call if a server is available.

**Note:** Authenticated RPC is not supported with the automatic binding method.

If the client stub cannot find a server to run the operation, it reports this by returning the status code **rpc_s_no_more_bindings** in the **comm_status** parameter, or by raising the exception **rpc_x_no_more_bindings** if the operation does not use the **comm_status** attribute for error reporting. If a binding breaks, the RPC runtime starts its search at the directory service entry following the one where the binding broke. Even if a server earlier in the list becomes available, it is not treated as a candidate for binding. After the RPC runtime tries each server in the list, it reinitializes the list of server candidates and tries again. If the second attempt is unsuccessful, the RPC runtime reports the status code, **rpc_s_no_more_bindings**. The next call on an operation in the interface starts from the top of the list when looking for a server to bind to.

The **auto_handle** attribute can occur at most once in the ACF.

If an interface uses the **auto_handle** attribute, the presence of a binding handle or context handle parameter in an operation overrides **auto_handle** for that operation.

The **auto_handle** attribute declaration has the following syntax. (See the example at the end of this section.)

- For an interface:

**[auto_handle] interface** *interface_name*

You cannot use **auto_handle** if you use **implicit_handle** or if you use **explicit_handle** in the interface header. You also cannot use **auto_handle** if you use the **encode** or **decode** ACF attributes.

**Example Using the auto_handle Attribute**

**ACF**

```
[auto_handle] interface math_1
{
}
```

**IDL File**

```
[uuid(b3c86900-2d27-11c9-ab09-08002b0ecef1)]
interface math_1
{
/* This operation has no handle parameter,
 * therefore, uses automatic binding.
 */
long add([in] long a,
         [in] long b);

/*
 * This operation has an explicit handle parameter, h,
 * that overrides the [auto_handle] ACF attribute.
 * Explicit handles also override [implicit_handle].
 */
long subtract ([in] handle_t h,
               [in] long a,
               [in] long b);
}
```

## The explicit_handle Attribute

This attribute allows the application program to manage the binding to the server. The **explicit_handle** attribute indicates that a binding handle is passed to the runtime as an operation parameter.

The **explicit_handle** attribute has the following syntax. (See the example at the end of this section.)

- For an interface:

  **[explicit_handle] interface** *interface_name*

- For an operation:

  **[explicit_handle]** *operation_name* **(** *[ parameter_list ]* **) ;**

When used as an ACF interface attribute, the **explicit_handle** attribute applies to all operations in the IDL file. When used as an ACF operation attribute, this attribute applies to only the operation you specify.

If you use the **explicit_handle** attribute as an ACF interface attribute, you must not use the **auto_handle** or **implicit_handle** attributes. Also, you cannot use the **encode** or **decode** ACF attributes if you use **explicit_handle**.

Using the **explicit_handle** attribute on an interface or operation has no effect on operations in IDL that have explicit binding information in their parameter lists.

**Example Using the explicit_handle Attribute**

**ACF**

```
[explicit_handle] interface math_2
 {

 /* This causes the operation, as called by the client, to have the
  * parameter handle_t IDL_handle, at the start of the parameter
  * list, before the parameters specified here in the IDL file.
  */
  }
```

**IDL File**

```
[uuid(41ce5b80-0ba7-11ca-87ba-08002b111685)]
interface math_2
{
long add([in] long a,
         [in] long b);
}
```

## The implicit_handle Attribute

This attribute allows the application program to manage the binding to the server. You specify the data type and name of the handle variable as part of the **implicit_handle** attribute. The **implicit_handle** attribute informs the compiler of the name and type of the global variable through which the binding handle is implicitly passed to the client stub. A variable of this type and name is defined in the client stub code, and the application initializes the variable before making a call to this interface.

The **implicit_handle** attribute declaration has the following syntax. (See the example at the end of this section.)

- For an interface:

  **[implicit_handle (** *handle_type handle_name* **)] interface** *interface_name*

If an interface uses the **implicit_handle** attribute, the presence of a binding handle or **in** or **in,out** context handle parameter in an operation overrides the implicit handle for that operation.

The **implicit_handle** attribute can occur at most once in the ACF.

You cannot use the **implicit_handle** attribute if you are using the **auto_handle** attribute or the **explicit_handle** attribute as an interface attribute. You also cannot use **implicit_handle** if you use the **encode** or **decode** ACF attributes.

If the type in the **implicit_handle** clause is not **handle_t**, then it is treated as if it has the **handle** attribute. For a description of the **handle** attribute see Chapter 11, “Interface Definition Language” on page 221.

The ACF in the following example changes the **math_3** interface to use an implicit handle.

**Example Using the implicit_handle Attribute**

**ACF**

```
[implicit_handle(user_handle_t global_handle)] interface math_3
{
/*
 * Since user_handle_t is not a type defined in the IDL, you
 * must specify an include file that contains the definition
 */
include "user_handle_t_def";
}
```

**IDL File**

```
[uuid(a01d0280-2d27-11c9-9fd3-08002b0ecef1)]
interface math_3
{
long add([in] long a,
         [in] long b);
}
```

## The comm_status and fault_status Attributes

The **comm_status** and **fault_status** attributes cause the status code of any communications failure or server runtime failure that occurs in an RPC to be stored in a parameter or returned as an operation result, instead of being raised to the client user code as an exception.

The **comm_status** attribute causes communications failures to be reported through a specified parameter. The **fault_status** attribute causes server runtime failures to be reported through a specified parameter. Applying both attributes causes all remote and communications failures to be reported through status. Any local exception caused by an error during marshalling, correctness checking performed by the client stubs, or an error in application routines continues to be returned as an exception.

The **comm_status** and **fault_status** attributes have the following syntax.

- For an operation:

  **[comm_status | fault_status]** *operation_name* **(***[parameter_list]***);**

- For a parameter:

  *operation_name* **([comm_status | fault_status]** *parameter_name***);**

**Note:** You can apply one of each attribute to the same operation and possibly the parameter at the same time. Separate the attributes with a comma.

If the parameter named in a **comm_status** or **fault_status** attribute is in the parameter list for the operation in the IDL file, it must have the **out** attribute in the IDL file. (Additional ACF parameters do not have **in** and **out** directional attributes.)

If the status attribute occurs on the operation, the returned value result must be defined as type **error_status_t** in the IDL file. If an error occurs during the operation, the error code is returned as the operation result. If the operation completes successfully, the value returned to the client is the value returned by the manager code.

**Note:** The **error_status_t** type is equivalent to **unsigned32**, which is the data type used by the RPC runtime for an error status. The status code **error_status_ok** is equivalent to **rpc_s_ok**, which is the RPC runtime success status code.

If the status attribute occurs on a parameter, the parameter name does not have to be defined in the IDL file, although it can be. Note the following:

- If the parameter name is one used in the IDL file, that parameter must be an output parameter of type **error_status_t**. If the operation completes successfully, the value of this parameter is the value returned by the manager code.
- If the parameter name is different from any name defined within the operation definition in the IDL file, then the IDL compiler creates an extra output parameter of type **error_status_t** in your application code after the last parameter defined in the IDL file. In a successfully completed remote call, the extra parameter has the value **error_status_ok**.

In either case, if an error occurs during the remote call, the error code is returned to the parameter that has the status attribute. *z/OS DCE Messages and Codes* describes the status codes.

If you define both additional **comm_status** and additional **fault_status** parameters, they are automatically added at the end of the procedure declaration in the order of specification in the ACF.

## The code and nocode Attributes

The **code** and **nocode** attributes allow you to control which operations in the IDL file have client stub code generated for them by the compiler. These attributes affect only the generation of a client stub; they have no effect when generating the server stub.

The **code** and **nocode** attributes have the following syntax. (See the examples at the end of this section.)

- For an interface:

  **[code | nocode] interface** *interface_name*

- For an operation:

  **[code | nocode]** *operation_name* **(***[parameter_list]***);**

When you specify **nocode** as an attribute on an ACF interface, stub code is not generated for the operations in the corresponding IDL interface unless you also specify **code** for the particular operations for which you want stub code generated. Similarly, when you specify **code** (the default) as an attribute on an ACF interface, stub code is generated for the operations in the corresponding IDL interface unless you also specify **nocode** for the particular operations for which you do not want stub code generated.

Do *not* use **nocode** on any of the operations if the compiler is generating only server stub code, because it has no effect. Server stubs must always contain generated code for all operations.

In the following example, the IDL compiler generates client stub code for the operations **open**, **read**, and **close**, but not for the operation **write**. An alternative method for specifying the same behavior is to use **[nocode] write()** in the ACF.

**Example Using the code and no_code Attributes**

**ACF**

```
[nocode,auto_handle] interface open_read_close
{
[code] open();
[code] read();
[code] close();
}
```

Note that at least one operation in your ACF file should have the **[code]** attribute (the default) specified. That is, you should not specify all operations in your IDL file as **[nocode]** or else the IDL compiler will not generate executable client stub code for any operations. This makes the client stub useless for those operations.

**IDL File**

```
[uuid(2166d580-0c69-11ca-811d-08002b111685)]
interface open_read_close
{
void open (...);
void read (...);
void write (...);
void close (...);
}
```

## The represent_as Attribute

This attribute associates a local data type that your application code uses with a data type defined in the IDL file. Use of the **represent_as** attribute means that during marshalling and unmarshalling, conversions occur between the data type used by the application code, and the data type specified in the IDL.

The **represent_as** attribute has the following syntax (See the example at the end of this section.)

**typedef [represent_as (***local_type_name***)]** *net_type_name***;**

The *local_type_name* is the local data type that the application code uses. You can define it in the IDL file or in an application header file. If you do not define it in the IDL file, use the **include** statement in the ACF to make its definition available to the stubs.

The *net_type_name* is the data type that is defined in the IDL file.

The **represent_as** attribute can appear at most once in a **typedef** declaration in an ACF.

If you use the **represent_as** attribute, you must write routines that perform the conversions between the local and network types, and routines that release the memory storage used to hold the converted data. The conversion routines are part of your application code.

The suffix for the routine names, the function of each, and where they are used (client or server) are shown in the following list:

| | |
|---|---|
| **_from_local** | Allocates storage instance of the network type and converts from the local type to the network type (used for client and server). |
| **_to_local** | Converts from the network type to the local type (used for client and server). |
| **_free_inst** | Frees storage instance used for the network type (used by client and server). |
| **_free_local** | Frees storage used by the server for the local type (used in server). This routine frees any object pointed to by its argument, but does not attempt to free the argument itself. |

Suppose that the **represent_as** attribute is applied to either the type of a parameter or to a component of a parameter, and that the parameter has the **out** or **in,out** attribute. Then the **_free_local** routine will be called automatically for the data item that has the type to which the **represent_as** attribute was applied.

Suppose that the **represent_as** attribute is applied to the type of a parameter and that the parameter has only the **in** attribute. Then the **_free_local** routine will be called automatically.

Finally, suppose that the **represent_as** attribute is applied to the type of a component of a parameter, and that the parameter has only the **in** attribute. Then the **_free_local** routine will not be called automatically for the component. The manager application code must release any resources that the component uses, possibly by explicitly calling the **_free_local** routine.

Append the suffix of the routine name to the *net_type_name*. The syntax for these routines is as follows:

**void** *net_type_name***_from_local (**
**(***local_type_name* ***),**
**(***net_type_name* **** ))**

**void** *net_type_name***_to_local (**
**(***net_type_name* *** ),**
**(***local_type_name* ***))**

**void** *net_type_name***_free_inst ((***net_type_name* ***))**

**void** *net_type_name_***free_local** **((***local_type_name* ***))**

**Example Using the represent_as Attribute**

**ACF**

```
[auto_handle] interface phonedir
{
/*
 * You must specify an included file that contains the
 * definition of my_dir_t.
 */
include "user_types";

/*
 * The application code wants to pass data type my_dir_t
 * rather than dir_t. The [represent_as] clause allows
 * this, and you must supply routines to convert dir_t
 * to/from my_dir_t.
 */
typedef [represent_as(my_dir_t)] dir_t;
}
```

**IDL File**

```
[uuid(06a12100-2d26-11c9-aa24-08002b0ecef1)]
interface phonedir
{
typedef struct
    {
    short int  area_code;
    long int   phone_num;
    char       last_name[20];
    char       first_name[15];
    char       city[20];
    } dir_t;
void add ([in] dir_t *info);
void lookup ([in] char city[20],
             [in] char last_name[20],
             [in] char first_name[15],
             [out] dir_t *info);
void delete ([in] dir_t *info);
}
```

## The enable_allocate Attribute

The **enable_allocate** attribute on an operation causes the server stub to initialize the **rpc_ss_allocate()** routine. The **rpc_ss_allocate()** routine requires initialization of its environment before it can be called. The server stub automatically initializes (enables) **rpc_ss_allocate()** if the operation uses either full pointers, or a type with the **represent_as** attribute. If the operation does not meet either of these conditions, but the manager application code needs to make use of the **rpc_ss_allocate()** and **rpc_ss_free()** routines, then use the **enable_allocate** attribute to force the stub code to enable.

The **enable_allocate** attribute has the following syntax:

- For an operation:

  **[enable_allocate]** *operation_name* **(***[parameter_list]***);**

**Example Using the enable_allocate Attribute**

**ACF**

```
[auto_handle] interface phonedir
{
[enable_allocate] lookup ();
}
```

**IDL File**

```
[uuid(06a12100-2d26-11c9-aa24-08002b0ecef1)]
interface phonedir
{
typedef struct
    {
    short int  area_code;
    long int   phone_num;
    char       last_name[20];
    char       first_name[15];
    char       city[20];
    } dir_t;
void add ([in] dir_t *info);
void lookup ([in] char city[20],
             [in] char last_name[20],
             [in] char first_name[15],
             [out] dir_t *info);
void delete ([in] dir_t *info);
}
```

## The heap Attribute

This attribute specifies that the server stub's copy of a parameter or of all parameters of a specified type is allocated in heap memory, rather than on the stack.

The **heap** attribute has the following syntax. (See the example at the end of this section.)

- For a type:

  **typedef [heap]** *type_name***;**

- For a parameter:

  *operation_name* **([heap]** *parameter_name***);**

Any identifier occurring as a parameter name within an operation declaration in the ACF must also be a parameter name within the corresponding operation declaration in the IDL.

The **heap** attribute is ignored for pipes, context handles, and scalars.

**Example Using the heap Attribute**

**ACF**

```
[auto_handle] interface galaxies
{
typedef [heap] big_array;
}
```

**IDL File**

```
[uuid(e61de280-0d0b-11ca-6145-08002b111685)]
interface galaxies
{
typedef long big_array[1000];
}
```

## The extern_exceptions Attribute

By default, the IDL compiler declares and initializes all exceptions listed in an **exceptions** interface attribute in the stub code that it generates. You can use the **extern_exceptions** attribute to override this behavior; the **extern_exceptions** attribute allows you to specify one or more exceptions listed in the exceptions interface attribute that you do not want the IDL-generated stub code to declare.

The **extern_exceptions** attribute has the following syntax. (See the example at the end of this section.)

**[extern_exceptions (***exception_name [,exception_name]***...)] interface** *interface_name*

The **extern_exceptions** attribute indicates that the specified exceptions are defined and initialized in some other external manner before calling the **extern_exceptions** attribute. They may be predefined exceptions (such as **exc_e_exquota**) that were provided by another interface, or exceptions that are defined and initialized explicitly by the application itself.

### Example Using the extern_exceptions Attribute

In the following example, the exception named in the list in the **extern_exceptions** attribute in the ACF is not defined or initialized in the IDL-generated stub code. All of the other exceptions listed in the **exceptions** interface attribute are defined and initialized in the generated stub.

**ACF**

```
[extern_exceptions(exc_e_quota)] interface binop {}
 /*
 *The exc_e_exquota exception is a predefined exception
 *(provided in exc_handling.h) and so does not need
 *to be declared and initialized in the IDL-generated stub.
 */
```

**IDL File**

```
[uuid(06255501-08af-11cb-8c4f-08002b13d56d),
version (1.1),
  exceptions (
      exc_e_exquota,
      binop_e_aborted,
      binop_e_too_busy,
      binop_e_shutdown)
] interface binop
  {
      long binop_add(
           [in] long a,
           [in] long b
           );
   }
```

## The encode and decode Attributes

The **encode** and **decode** attributes are used in conjunction with IDL encoding services routines (**idl_es***) to enable RPC applications to encode data types in input parameters into a byte stream and decode data types in output parameters from a byte stream without invoking the RPC runtime. Encoding and decoding operations are analogous to marshalling and unmarshalling, except that the data is stored locally, and is not transmitted over the network.

The stubs that perform encoding or decoding operations are different from the stubs that perform RPC operations. The ACF attributes **encode** and **decode** direct the IDL compiler to generate encoding or decoding stubs for operations defined in a corresponding IDL interface rather than generating RPC stubs for those operations.

The **encode** and **decode** attributes have the following syntax. (See the example at the end of this section.)

For an interface:

**[encode]** | **[decode]** | **[encode,decode] interface** *interface_name*

For an operation:

**[encode]** | **[decode]** | **[encode,decode]** *operation_name* **(***[parameter_list]***);**

When used as an ACF interface attribute, the **encode** and **decode** attributes apply to all operations defined in the corresponding IDL file. When used as an ACF operation attribute, **encode** and **decode** apply only to the operation you specify. If you apply the **encode** or **decode** attribute to an ACF interface or operation, you must not use the **auto_handle** or the **implicit_handle** ACF attributes.

When you apply the **encode** or **decode** attribute to an operation, the IDL compiler generates IDL encoding services stubs that support encoding or decoding, depending on the attribute used, in the client stub code, and does not generate stub code for the operation in the server stub. To generate an IDL encoding services stub that supports both encoding and decoding, apply both attributes to the operation.

If you apply the **encode** or **decode** attribute to all of the operations in an interface, no server stub is generated. If you apply the **encode** and **decode** attributes to some, but not all, of the operations in an interface, the stubs for the operations that do not have the **encode** and **decode** attributes applied to them are generated as RPC stubs.

When data encoding takes place, only the operation's **in** parameters provide data for the encoding. When data decoding takes place, the decoded data is delivered only to the operation's **out** parameters.

If data is being both encoded and decoded, you generally declare all of the operation's parameters to be **in**,**out**. However, you can encode data using the **in** parameters of one operation, and decode it using the **out** parameters of another operation if the types and order of the **in** and **out** parameters are the same. For equivalence, the IDL encoding services treat a function result as an **out** parameter that appears after all other **out** parameters.

In the following example, the IDL compiler generates IDL encoding services stub code for the **in_array_op1**, **out_array_op1**, and **array_op2** operations, but not for the **array_op3** operation. The stub code generated for the **in_array_op1** operation supports encoding, the stub code generated for the **out_array_op1** operation supports decoding, and the stub code generated for the **array_op2** operation supports both encoding and decoding. The stub code generated for the **array_op3** is an RPC client stub. For further information on using the IDL encoding services, see “Creating Portable Data Using the IDL Encoding Services” on page 211 and the *z/OS DCE Application Development Reference*.

### Example Using the encode and decode Attributes

**ACF**

```
interface es_array
{
    [encode] in_array_op1();
    [decode] out_array_op1();
    [encode, decode] array_op2();
}
```

**IDL File**

```
[uuid(20aac780-5398-11c9-b996-08002b13d56d), version(0)]
interface es_array
{
    void in_array_op1([in] handle_t h, [in] long arr[100]);
    void out_array_op1([in] handle_t h, [out] long arr[100]);
    void array_op2([in] handle_t h, [in,out] long big[100]);
    void array_op3([in] handle_t h, [in,out] long big[100]);
}
```

## The cs_char Attribute

The **cs_char** attribute is intended for use in internationalized RPC applications. It is used in conjunction with the **cs_stag**, **cs_drtag**, **cs_rtag** and **cs_tag_rtn** attributes and the DCE RPC routines for automatic code set conversion to provide RPC applications with a mechanism for ensuring character and code set interoperability between clients and servers transferring international (non-PCS) characters.

The **cs_char** attribute is very similar in function to the **represent_as** attribute: it associates a local data type that your application code uses with a data type defined in the IDL file. The **cs_char** attribute permits the application code to use the local data type for international character data, and converts between the local data type and the format specified in the IDL file when transferring international characters over the network. The **cs_char** ACF attribute permits the conversion of characters, arrays of characters, and strings of characters between the format in which the application code requires them and the format in which they are transmitted over the network.

As with **represent_as**, use of the **cs_char** attribute means that during marshalling and unmarshalling, conversions occur between the data type that the application code is using and the data type specified in the IDL. In the case of **cs_char**, the local data type is automatically converted between the local data type in the local code set encoding and the **idl_byte** data type in the network code set encoding. The network code set is the code set encoding that the application code, through the use of code set conversion routines, has selected to use when transmitting the international characters over the network.

The **cs_char** attribute differs from the **[transmit_as]** attribute in that it does not affect the network contract between the client and server. It differs from the **[represent_as]** attribute in that multiple data items (for example, the characters of an array or string) can be converted with a single stub call to a code set conversion routine, and that the conversion can modify array size and data limit information between what is transmitted over the network and what is used by application code.

The **cs_char** attribute has the following syntax. (See the examples at the end of this section.)

**typedef [cs_char (***local_type_name***)]** *net_type_name;*

The *local_type_name* is the local data type that the application code uses. You can define it in the IDL file or in an application header file. If you do not define it in the IDL file, use the **include** statement in the ACF to make its definition available to the stubs.

The *net_type_name* is the data type that is defined in the IDL file. When used with the **cs_char** attribute, this data type is always **byte** in the IDL file.

If you use the **cs_char** attribute, you must write the following stub support routines for each local type that you define:

- Routines that check the buffer storage requirements for international character data to be converted to determine whether or not more buffer space needs to be allocated to hold the converted data
- Routines to perform conversion between local and network code sets

The suffix for the routine names, the function of each, and where they are used (client or server) appear in the following list:

| | |
|---|---|
| *local_type_name*_**net_size()** | Calculates the necessary buffer size for code set conversion from a local code set to a network code set. Client and server stubs call this routine before they marshall any international character data. |
| *local_type_name*_**local_size()** | Calculates the necessary buffer size for code set conversion from a network code set to a local code set. Client and server stubs call this routine before they unmarshall any international character data. |
| *local_type_name*_**to_netcs()** | Converts international character data from a local code set to a network code set. Client and server stubs call this routine before they marshall any international character data. |
| *local_type_name*_**from_netcs()** | Converts international character data from a network code set to a local code set. Client and server stubs call this routine after they unmarshall any international character data. |

You specify the name for the local data type in the *local_type_name* portion of the function name. The name that you specify cannot exceed 20 characters, because the entire generated name must not exceed the 31-character limit for C identifiers.

For each piece of international character data being marshalled, the **_net_size** and **_to_netcs** routines are called once each. For each piece of international character data being unmarshalled, the **_local_size** and **_from_netcs** routines are called once each.

DCE RPC provides buffer sizing and code set conversion routines for the **cs_byte** data types (the **cs_byte** type is equivalent to the **byte** type). If they meet the needs of your application, you can use these RPC routines (**cs_byte_*** ) instead of providing your own routines.

If you do provide your own routines for buffer sizing and code set conversion, they must follow a specific signature. See the *z/OS DCE Application Development Reference* for a complete description of the required signatures for the **cs_byte_** routines.

**Restrictions** When international character data is to be unmarshalled, a stub needs to have received a description of the codeset being used before it receives the data. For this reason, the **cs_char** attribute cannot be applied to the base type of a pipe, or to a type used in constructing the base type of a pipe.**:** The **cs_char** attribute also cannot be applied to a type if there is an array that has this type as a base type and the array has more than one dimension, or if the attributes **min_is**, **max_is**, **first_is**, **last_is**, or **string** have been applied to the array. As a result, all instances of the type to which **cs_char** has been applied must be scalars or one-dimensional arrays. Only the **length_is** and/or **size_is** attributes can be applied to these arrays.

The following restrictions apply to the use of variables that appear in array attributes:

- Any parameter that is referenced by a **size_is** or **length_is** attribute of an array parameter whose base type has the **cs_char** attribute cannot be referenced by any attribute of an array parameter whose base type does not have the **cs_char** attribute.
- Any structure field that is referenced by a **size_is** or **length_is** attribute of an array field whose base type has the **cs_char** attribute cannot be referenced by any attribute of an array field whose base type does not have the **cs_char** attribute.

The **cs_char** attribute cannot interact with the **transmit_as** or **represent_as** attributes. This restriction imposes the following rules:

- The **cs_char** attribute cannot be applied to a type that has the **transmit_as** attribute, nor can it be applied to a type in whose definition a type with the **transmit_as** attribute is used.
- The **cs_char** attribute cannot be applied to a type that has the **represent_as** attribute, nor can it be applied to a type in whose definition a type with the **represent_as** attribute is used.
- The **cs_char** attribute cannot be applied to the transmitted type specified in a **transmit_as** attribute or to any type used in defining such a transmitted type.

The **cs_char** attribute cannot be applied to any type belonging to the referent of a pointer with a **max_is** or **size_is** attribute. It also cannot be applied to the base type of an array parameter that has the **unique** or **ptr** attribute applied to it.

An application that uses the **cs_char** ACF attribute cannot use the IDL encoding services **encode** and **decode** ACF attributes.

### Examples Using the cs_char Attribute

Arrays of **cs_char** can be fixed, varying, conformant, or conformant varying. The treatment of a scalar **cs_char** is similar to that of a fixed array of one element. The following examples show the relationship between IDL declarations and declarations in the generated header file when the **cs_char** attribute has been applied. The examples assume that the ACF contains the type definition:

```
typedef [cs_char(ltype)] my_byte;
```

For a fixed array, if the IDL file contains:

```
        typedef struct {
            my_byte fixed_array[80];
        } fixed_struct;
```

the declaration generated in the header file is:

```
        typedef struct {
            ltype fixed_array[80];
        } fixed_struct;
```

The number of array elements in the local and network representations of the data must be the same as the array size stated in the IDL.

For a varying array, if the IDL file contains:

```
        typedef struct {
            long l;
            [length_is(l)] my_byte varying_array[80];
        } varying_struct;
```

the declaration generated in the header file is:

```
        typedef struct {
            idl_long_int l;
            ltype varying_array[80];
        } varying_struct;
```

Neither the number of array elements in the local representation nor the number of array elements in the network representation may exceed the array size in the IDL.

For a conformant array, if the IDL file contains:

```
        typedef struct {
            long s;
            [size_is(s)] my_byte conf_array[];
        } conf_struct;
```

the declaration generated in the header file is:

```
        typedef struct {
            idl_long_int s;
            ltype conf_array[1];
        } conf_struct;
```

The number of array elements in the local representation and the number of array elements in the network representation need not be the same. The conversions between these numbers are done in the **_net_size** and **_local_size** routines.

For a conformant varying array, if the IDL file contains:

```
        typedef struct {
            long s;
            long l;
            [size_is(s), length_is(l)] my_byte open_array[];
        } open_struct;
```

the declaration generated in the header file is:

```
        typedef struct {
            idl_long_int s;
            idl_long_int l;
            ltype open_array[1];
        } open_struct;
```

The maximum number of array elements in the local representation and the maximum number of array elements in the network representation need not be the same. The conversions between these numbers are done in the **_net_size** and **_local_size** routines.

For fixed or varying arrays, the size of the storage available to hold the local data is determined by the array size specified in the IDL and the local type specified in the **cs_char** attribute. The array size is fixed and cannot be modified during the RPC marshalling and unmarshalling.

For fixed arrays, the number of bytes of data on the client, the server, and the network must be exactly equal to the number defined in the IDL file. Following are additional restrictions for fixed arrays:

- The number of array elements in the local (client and server) and network representations of the data must be the same as the array size defined in the IDL.
- Because the array size is the input length used by the code set conversion, the complete array must be populated with valid data.
- Customized stub buffer sizing routines and code set conversion routines must be written if the application requires the **idl_cs_in_place_convert** conversion type to be used. The routines provided by z/OS DCE RPC do not support the **idl_cs_in_place_convert** conversion type.
- Customized stub tag-setting routines may be written or the DCE RPC tag-setting routine **rpc_cs_get_tags** may be invoked to set the appropriate tag values for transmitting the data over the network. The application programmer and application administrator must ensure that the code set conversion between server and client will not result in data expansion or contraction.
- Customized character and code sets compatibility evaluation routines may be written. z/OS DCE RPC **rpc_cs_eval_with_universal()** must not be invoked because universal conversion may cause data

expansion. **rpc_cs_eval_without_universal()** may be used, but remember that the conversion model used by this routine is: RMIR first, then SMIR, then CMIR. When using this routine, the application programmer and application administrator must ensure that conversions can be performed without data expansion or contraction.

For a varying array, neither the number of array elements in the local representation nor the number of array elements in the network representation may exceed the array size in the IDL.

Restrictions similar to those for fixed arrays also apply to varying arrays. The value of *length_is* is the input length used by the code set conversion routine. Expansion and contraction of data is allowed within the array size defined in the IDL file.

Conformant or conformant varying arrays can be used without restrictions because they are designed to allow the data expansion and contraction which can occur during code set conversion. You must determine the transformations between local storage size and network storage size with reference to the characters being transmitted or received. Where a variable-width character set is in use, this means making the most conservative assumption about the size of the data.

## The cs_stag, cs_drtag, and cs_rtag Attributes

The **cs_stag**, **cs_drtag** and **cs_rtag** attributes are used in conjunction with the **cs_char** and (optionally) the **cs_tag_rtn** attributes and DCE RPC routines for automatic code set conversion to provide internationalized RPC applications with a mechanism to ensure character and code set interoperability between clients and servers handling international character data.

The **cs_stag**, **cs_drtag** and **cs_rtag** attributes are parameter ACF attributes that correspond to the sending tag, desired receiving tag, and receiving tag parameters defined in operations in the IDL file that handle international character data. These operation parameters tag international characters being passed in the operation's input and output parameters with code set identifying information. The **cs_stag**, **cs_drtag** and **cs_rtag** ACF parameter attributes declare the tag parameters in the corresponding operation definition to be special code set parameters.

The **cs_stag** attribute has the following syntax:

*operation_name* **([cs_stag]** *parameter_name***);**

The **cs_stag** attribute identifies the code set used when the client sends international characters to the server. Operations defined in the IDL file that specify international characters in **in** parameters must use the **cs_stag** attribute in the associated ACF.

The **cs_drtag** attribute has the following syntax:

*operation_name* **([cs_drtag]** *parameter_name***);**

The **cs_drtag** attribute identifies the code set the client would like the server to use when returning international characters.

The **cs_rtag** attribute has the following syntax:

*operation_name* **([cs_rtag]** *parameter_name***);**

The **cs_rtag** attribute identifies the code set that is actually used when the server sends international characters to the client. Operations defined in the IDL file that specify international characters in **out** parameters must apply the **cs_rtag** attribute in the associated ACF.

### Example Using the cs_stag, cs_drtag, and cs_rtag Attributes

Here is an example ACF for an IDL file in which the operation **my_op** has the tag parameters **my_stag**, **my_drtag**, and **my_rtag**, whose types are either **unsigned long** or **[ref] unsigned long**.

```
my_op( [cs_stag] my_stag, [cs_drtag] my_drtag, [cs_rtag] my_rtag);
```

For more information about the **cs_stag**, **cs_drtag**, and **cs_rtag** ACF attributes and their use in internationalized RPC applications, see Chapter 9, "Writing Internationalized RPC Applications" on page 143.

## The cs_tag_rtn Attribute

The **cs_tag_rtn** attribute is an ACF attribute for use in RPC applications that handle international character data. This attribute specifies the name of a routine that the client and server stubs will call to set an operation's code set tag parameters to specific code set values. The **cs_tag_rtn** attribute is an optional ACF attribute that you can use to provide code set tag transparency for callers of your interface's operations. If an operation that transfers international character data has the **cs_tag_rtn** attribute applied to it in the corresponding ACF, the code set tag parameters will not appear in the operation's definition within the generated header file. If the **cs_tag_rtn** attribute is not used, the operation's caller must provide appropriate values to the operation's code set tag parameters before international character data is marshalled.

The **cs_tag_rtn** attribute has the following syntax. (See the example at the end of this section.)

For an interface:

**[cs_tag_rtn (***tag_set_routine***)] interface** *interface_name*

For an operation:

**[cs_tag_rtn (***tag_set_routine***)]** *operation_name* **(***[parameter_list]***);**

When used as an ACF interface attribute, the **cs_tag_rtn** attribute applies to all operations defined in the corresponding IDL file. When used as an ACF operation attribute, the **cs_tag_rtn** attribute applies only to the operation you specify.

The *tag_set_routine* is the name of the stub support routine that the client and server stubs will call to set the operation's code set tag parameters. The IDL compiler will generate a function prototype for *tag_set_routine* in the generated header file.

Applications can specify the DCE RPC tag-setting routine **rpc_cs_get_tags()**, if it meets their applications' needs, or they can write their own tag-setting routines. The routine name must be distinct from any type name, procedure name, constant name, or enumeration name appearing in the interface definition. It must also have a specific calling signature. See the description of the **rpc_cs_get_tags()** routine in the *z/OS DCE Application Development Reference* for a complete description of the required routine signature.

When the tag-setting routine is called from a client stub, it is called before any **in** parameters are marshalled. When called from a server stub, it is called before any **out** parameters are marshalled. For more information on the **cs_tag_rtn** attribute and its use in internationalized RPC applications, see Chapter 9, "Writing Internationalized RPC Applications" on page 143.

### Example Using the cs_tag_rtn Attribute

As shown in the following example, the **cs_tag_rtn** attribute is used in conjunction with the **cs_char**, **cs_stag**, **cs_drtag** and **cs_rtag** ACF attributes. In the example, the stub generated for **a_op** will call the tag-setting routine **set_tags** to set the code set tag parameters to specific values before any data is marshalled. For **b_op**, it is the responsibility of the operation's caller to ensure that the code set tag parameters are set correctly before any data is marshalled.

**IDL File**

```
typedef byte my_byte;

void a_op(
    [in] unsigned long stag,
    [in] unsigned long drtag,
    [out] unsigned long *p_rtag,
    [in] long s,
    [in, out] long *p_l,
    [in, out, size_is(s), length_is(*p_l)] my_byte a[]
);

void b_op(
    [in] unsigned long stag,
    [in] unsigned long drtag,
    [out] unsigned long *p_rtag,
    [in] long s,
    [in, out] long *p_l,
    [in, out, size_is(s), length_is(*p_l)] my_byte a[]
);
```

**ACF**

```
typedef [cs_char(ltype)] my_byte;

[cs_tag_rtn(set_tags)] a_op( [cs_stag] stag,
                         [cs_drtag] drtag,
                         [cs_rtag] p_rtag );

                     b_op( [cs_stag] stag,
                           [cs_drtag] drtag,
                           [cs_rtag] p_rtag );
```

**Generated Header File**

```
typedef byte my_byte;

void a_op(
    /* [in] */ idl_long_int s,
    /* [in, out] */ idl_long_int *p_l,
    /* [in, out, size_is(s), length_is(*p_l)] */ ltype a[]
);

void b_op(
    /* [in] */ idl_ulong_int stag,
    /* [in] */ idl_ulong_ing drtag,
    /* [out] */ idl_ulong_int *p_rtag,
    /* [in] */ idl_long_int s,
    /* [in, out] */ idl_long_int *p_l,
    /* [in, out, size_is(s), length_is(*p_l)] */ ltype a[]
);
```

## The binding_callout Attribute

The **binding_callout** attribute permits you to specify the name of a routine that the client stub is to call automatically to modify a server binding handle before it initiates a remote procedure call. This attribute is intended for use by client applications that employ the automatic binding method through the **auto_handle** ACF interface attribute. In automatic binding, it is the client stub, rather than the client application code, that obtains the binding handle to the server. The **binding_callout** attribute allows a client application using automatic binding to modify the binding handle obtained by the client stub. Without this attribute, it is impossible for the client application to modify the binding handle before the client stub attempts to initiate a remote procedure call to the selected server.

Clients typically use this attribute to augment automatic binding handles with security context, for example, so that authenticated RPC is used between client and server.

The **binding_callout** attribute has the following syntax.

**[binding_callout (***routine_name***)] interface** *interface_name*

The *routine_name* specifies the name of a binding callout routine that the client stub will call to modify the server binding handle before initiating the remote procedure call to the server. The IDL compiler will generate a function prototype for *routine_name* in the generated header file.

You can specify the name of a routine that you supply, or you can specify the DCE RPC routine **rpc_ss_bind_authn_client()** to modify the binding handle, if it meets the needs of your application. See the *z/OS DCE Application Development Reference* for a description of the **rpc_ss_bind_authn_client()** routine.

The binding callout routine you provide must have a specific routine signature. See the description of **rpc_ss_bind_authn_client()** in the *z/OS DCE Application Development Reference* for information about the required routine signature.

The **binding_callout** attribute can occur at most once in the ACF and applies to all of the operations in the corresponding IDL file.

A binding callout routine should return the **error_status_ok** status code when it successfully modifies the binding handle or determines that no action is necessary. This status code causes the client stub to initiate the remote procedure call.

A binding callout routine can also return error status. If it does, the client stub does not initiate the remote procedure call. Instead, if the **auto_handle** attribute has been applied in the ACF, the client stub attempts to locate another server of the interface, and then calls the binding callout routine again. If **auto_handle** is not in use, the client stub invokes its normal error-handling logic. A binding callout routine for a client using **auto_handle** can return the status code **rpc_s_no_more_bindings** to prevent the client stub from searching for another server and instead invoking its error-handling logic immediately.

By default, the client stub handles an error condition by raising an exception. If a binding callout routine returns one of the **rpc_s_** status codes, the client stub raises a matching **rpc_x_** exception. However, if a binding callout routine returns any other type of status code, the client stub will most likely raise it as an unknown status exception.

If the **comm_status** parameter ACF attribute has been applied to an operation, the client stub handles an error condition by returning the error status value in the **comm_status** parameter. Consequently, a binding callout routine can return any error status value to the client application code if the **comm_status** attribute has been applied to the operation.

A binding callout routine can raise a user-defined exception rather than return a status code to report application-specific error conditions back to the client application code using exceptions.

## Summary of Attributes

Table 13 lists the attributes available for use in the ACF and where in the file they can be used.

*Table 13. Summary of the ACF Attributes*

| Attribute | Where Used |
|---|---|
| **auto_handle** | Interface header |
| **binding_callout** | Interface header |
| **code** | Interface header, operation |
| **comm_status** | Operation, parameter |
| **cs_char** | Type |
| **cs_drtag** | Parameter |
| **cs_rtag** | Parameter |
| **cs_stag** | Parameter |
| **cs_tag_rtn** | Operation, interface |
| **decode** | Operation, interface |
| **enable_allocate** | Operation |
| **encode** | Operation, interface |
| **explicit_handle** | Interface header, operation |
| **extern_exception** | Interface header |
| **fault_status** | Operation, parameter |
| **heap** | Type, parameter |
| **implicit_handle** | Interface header |
| **nocode** | Interface header, operation |
| **represent_as** | Type |

## ACF Grammar Synopsis

The syntax description in this section uses an extended Backus-Naur Form (BNF) to represent ACF grammar. Table 14 lists the symbols used in this section and their meanings:

*Table 14 (Page 1 of 4). Backus-Naur Format for the Attribute Configuration File Language*

| Number | Production Rule | | |
|---|---|---|---|
| 1 | *<acf_interface>* | ::= | *<acf_interface_header> <acf_interface_body>* |
| 2 | *<acf_interface_header>* | ::= | *<acf_interface_attr_list>* **interface** *<acf_interface_name>* |
| 3 | *<acf_interface_attr_list>* | ::= | **[** *<acf_interface_attrs>* **]**<br>\| ϕ |

*Table 14 (Page 2 of 4). Backus-Naur Format for the Attribute Configuration File Language*

| Number | Production Rule | | |
|---|---|---|---|
| 4 | *<acf_interface_attrs>* | ::= | *<acf_interface_attr>* |
| | | | \| *<acf_interface_attrs>* **,** *<acf_interface_attr>* |
| 5 | *<acf_interface_attr>* | ::= | *<acf_code_attr>* |
| | | | \| *<acf_nocode_attr>* |
| | | | \| *<acf_binding_callout_attr>* |
| | | | \| *<acf_explicit_handle_attr>* |
| | | | \| *<acf_inline_attr>* |
| | | | \| *<acf_outofline_attr>* |
| | | | \| *<acf_implicit_handle_attr>* |
| | | | \| *<acf_auto_handle_attr>* |
| | | | \| *<acf_extern_exceps_attr>* |
| | | | \| **decode** |
| | | | \| **encode** |
| 6 | *<acf_implicit_handle_attr>* | ::= | **implicit_handle (** *<acf_implicit_handle>* **)** |
| 7 | *<acf_implicit_handle>* | ::= | *<acf_impl_type>* *<acf_impl_name>* |
| 8 | *<acf_impl_type>* | ::= | *<acf_handle_type>* |
| | | | \| **IDENTIFIER** |
| 9 | *<acf_handle_type>* | ::= | **handle_t** |
| 10 | *<acf_impl_name>* | ::= | **IDENTIFIER** |
| 11 | *<acf_extern_exceps_attr>* | ::= | **extern_exceps (** |
| | | ::= | \| **extern_exceps** |
| 12 | *<acf_ext_excep_list>* | ::= | *<acf_ext_excep>* |
| | | : | \| *<acf_ext_excep_list>* **,** *<acf_ext_excep>* |
| 13 | *<acf_ext_excep>* | ::= | **IDENTIFIER** |
| 14 | *<acf_interface_name>* | ::= | **IDENTIFIER** |
| 15 | *<acf_interface_body>* | ::= | **{** *<acf_body_elements>* **}** |
| | | | \| **{ }** |
| | | | \| error |
| | | | \| error **}** |
| 16 | *<acf_body_elements>* | ::= | *<acf_body_element>* |
| | | | \| *<acf_body_elements>* *<acf_body_element>* |
| 17 | *<acf_body_element>* | ::= | *<acf_include>* **;** |
| | | | \| *<acf_type_declaration>* **;** |
| | | | \| *<acf_operation_declaration>* **;** |
| | | | \| error **;** |
| 18 | *<acf_include>* | ::= | **include** *<acf_include_list>* |
| | | | \| **include** error |
| 19 | *<acf_include_list>* | ::= | *<acf_include_name>* |
| | | | \| *<acf_include_list>* **,** *<acf_include_name>* |
| 20 | *<acf_include_name>* | ::= | **STRING** |

*Table 14 (Page 3 of 4). Backus-Naur Format for the Attribute Configuration File Language*

| Number | Production Rule | | |
|---|---|---|---|
| 21 | *<acf_type_declaration>* | ::= | **typedef** error |
| | | | \| **typedef** *<acf_type_attr_list>* *<acf_named_type_list>* |
| 22 | *<acf_named_type_list>* | ::= | *<acf_named_type>* |
| | | | \| *<acf_named_type_list>* **,** *<acf_named_type>* |
| 23 | *<acf_named_type>* | ::= | **IDENTIFIER** |
| 24 | *<acf_type_attr_list>* | ::= | **[** *<acf_rest_of_attr_list>* |
| | | | \| $\phi$ |
| 25 | *<acf_rest_of_attr_list>* | ::= | *<acf_type_attrs>* **]** |
| | | | \| error **;** |
| | | | \| error **]** |
| 26 | *<acf_type_attrs>* | ::= | *<acf_type_attr>* |
| | | | \| *<acf_type_attrs>* **,** *<acf_type_attr>* |
| 27 | *<acf_type_attr>* | ::= | *<acf_represent_attr>* |
| | | | \| *<acf_heap_attr>* |
| | | | \| *<acf_inline_attr>* |
| | | | \| *<acf_outofline_attr>* |
| 28 | *<acf_represent_attr>* | ::= | **represent_as (** *<acf_repr_type>* **)** |
| 29 | *<acf_repr_type>* | ::= | **IDENTIFIER** |
| 30 | *<acf_operation_declaration>* | ::= | *<acf_op_attr_list>* *<acf_operations>* |
| 31 | *<acf_operations>* | ::= | *<acf_operation>* |
| | | | \| *<acf_operations>* **,** *<acf_operation>* |
| 32 | *<acf_operation>* | ::= | **IDENTIFIER (** *<acf_parameter_list>* **)** |
| 33 | *<acf_op_attr_list>* | ::= | **[** *<acf_op_attrs>* **]** |
| | | | \| $\phi$ |
| 34 | *<acf_op_attrs>* | ::= | *<acf_op_attr>* |
| | | | \| *<acf_op_attrs>* **,** *<acf_op_attr>* |
| 35 | *<acf_op_attr>* | ::= | *<acf_commstat_attr>* |
| | | | \| *<acf_code_attr>* |
| | | | \| *<acf_nocode_attr>* |
| | | | \| *<acf_enable_allocate_attr>* |
| | | | \| *<acf_explicit_handle_attr>* |
| | | | \| *<acf_faultstat_attr>* |
| | | | \| **encode** |
| | | | \| **decode** |
| 36 | *<acf_binding_callout_attr>* | ::= | **binding_callout (** *<acf_binding_callout_name>* **)** |
| 37 | *<acf_binding_callout_name>* | ::= | **IDENTIFIER** |
| 38 | *<acf_parameter_list>* | ::= | *<acf_parameters>* |
| | | | \| $\phi$ |

Table 14 (Page 4 of 4). Backus-Naur Format for the Attribute Configuration File Language

| Number | Production Rule | | |
|---|---|---|---|
| 39 | *<acf_parameters>* | ::= | *<acf_parameter>* <br> \| *<acf_parameters>* **,** *<acf_parameter>* |
| 40 | *<acf_parameter>* | ::= | *<acf_param_attr_list>* **IDENTIFIER** |
| 41 | *<acf_param_attr_list>* | ::= | **[** *<acf_param_attrs>* **]** <br> \| ϕ |
| 42 | *<acf_param_attrs>* | ::= | *<acf_param_attr>* <br> \| *<acf_param_attrs>* **,** *<acf_param_attr>* |
| 43 | *<acf_param_attr>* | ::= | *<acf_commstat_attr>* <br> \| *<acf_faultstat_attr>* <br> \| *<acf_heap_attr>* <br> \| *<acf_inline_attr>* <br> \| *<acf_outofline_attr>* |
| 44 | *<acf_auto_handle_attr>* | ::= | **auto_handle** |
| 45 | *<acf_code_attr>* | ::= | **code** |
| 46 | *<acf_nocode_attr>* | ::= | **nocode_kw** |
| 47 | *<acf_enable_allocate_attr>* | ::= | **enable_allocate** |
| 48 | *<acf_explicit_handle_attr>* | ::= | **explicit_handle** |
| 49 | *<acf_heap_attr>* | ::= | **heap** |
| 50 | *<acf_inline_attr>* | ::= | **in_line** |
| 51 | *<acf_outofline_attr>* | ::= | **out_of_line** |
| 52 | *<acf_commstat_attr>* | ::= | **comm_status** |
| 53 | *<acf_faultstat_attr>* | ::= | **fault_status** |

Table 15 is an index of the BNF production rules in Table 14 on page 306. Use Table 15 as an index to find where a production rule is defined and where it is used in Table 14.

Table 15 (Page 1 of 3). Cross Reference of the Backus-Naur Format for the Attribute Configuration File Language

| Production Rule | Defined | Used |
|---|---|---|
| *<acf_auto_handle_attr>* | 44 | *5* |
| *<acf_binding_callout_attr>* | 36 | |
| *<acf_binding_callout_name>* | 37 | |
| *<acf_body_element>* | 17 | *16* |
| *<acf_body_elements>* | 16 | *15, 16* |
| *<acf_code_attr>* | 45 | *5, 35* |
| *<acf_commstat_attr>* | 52 | *35, 43* |

*Table 15 (Page 2 of 3). Cross Reference of the Backus-Naur Format for the Attribute Configuration File Language*

| **Production Rule** | **Defined** | **Used** |
|---|---|---|
| *<acf_enable_allocate_attr>* | 47 | *35* |
| *<acf_explicit_handle_attr>* | 48 | *5, 35* |
| *<acf_ext_excep>* | 13 | *12* |
| *<acf_ext_excep_list>* | 12 | |
| *<acf_extern_exceps_attr>* | 11 | *5* |
| *<acf_faultstat_attr>* | 53 | *35, 43* |
| *<acf_handle_type>* | 9 | *8* |
| *<acf_heap_attr>* | 49 | *27, 43* |
| *<acf_impl_name>* | 10 | *7* |
| *<acf_impl_type>* | 8 | *7* |
| *<acf_implicit_handle>* | 7 | *6* |
| *<acf_implicit_handle_attr>* | 6 | *5* |
| *<acf_include>* | 18 | *17* |
| *<acf_include_list>* | 19 | *18, 19* |
| *<acf_include_name>* | 20 | *19* |
| *<acf_inline_attr>* | 50 | *5, 27, 43* |
| *<acf_interface>* | 1 | |
| *<acf_interface_attr>* | 5 | *4* |
| *<acf_interface_attr_list>* | 3 | *2* |
| *<acf_interface_attrs>* | 4 | *3, 4* |
| *<acf_interface_body>* | 15 | *1* |
| *<acf_interface_header>* | 2 | *1* |
| *<acf_interface_name>* | 14 | *2* |
| *<acf_named_type>* | 23 | *22* |
| *<acf_named_type_list>* | 22 | *21, 22* |
| *<acf_nocode_attr>* | 46 | *5, 35* |
| *<acf_op_attr>* | 35 | *34* |
| *<acf_op_attr_list>* | 33 | *30* |
| *<acf_op_attrs>* | 34 | *33, 34* |
| *<acf_operation>* | 32 | *31* |
| *<acf_operation_declaration>* | 30 | *17* |
| *<acf_operations>* | 31 | *30, 31* |
| *<acf_outofline_attr>* | 51 | *5, 27, 43* |
| *<acf_param_attr>* | 43 | *42* |
| *<acf_param_attr_list>* | 41 | *40* |
| *<acf_param_attrs>* | 42 | *41, 42* |
| *<acf_parameter>* | 40 | *39* |
| *<acf_parameter_list>* | 38 | *32* |
| *<acf_parameters>* | 39 | *38, 39* |
| *<acf_repr_type>* | 29 | *28* |
| *<acf_represent_attr>* | 28 | *27* |
| *<acf_rest_of_attr_list>* | 25 | *24* |

*Table 15 (Page 3 of 3). Cross Reference of the Backus-Naur Format for the Attribute Configuration File Language*

| **Production Rule** | **Defined** | **Used** |
|---|---|---|
| *<acf_type_attr>* | 27 | *26* |
| *<acf_type_attr_list>* | 24 | *21* |
| *<acf_type_attrs>* | 26 | *25, 26* |
| *<acf_type_declaration>* | 21 | *17* |

# Part 3. Using the DCE Threads APIs

This part shows you how to increase the performance of your distributed applications using the DCE Threads APIs. You are introduced to Threads concepts and shown several models for multithreaded programming. In addition, you are shown how to use the DCE Threads exception-handling interface to handle abnormal conditions in your applications. A comparison between DCE Threads concepts and z/OS multitasking will assist you in avoiding pitfalls caused by semantic differences.

Chapter 13, “Introduction to Multithreaded Programming” on page 315 to Chapter 17, “DCE Threads Example” on page 349 describes DCE Threads from OSF that is based on the POSIX 1003.4a, Draft 4 standard. The threads package that ships with UNIX System Services and Language Environment® is based on the POSIX 10003.4a, Draft 6 standard. This threads package, which is referred to as UNIX System Services Threads in this book, is fully described in *z/OS C/C++ Run-Time Library Reference*, SA22-7821. To enable interoperability with other DCE implementations, z/OS DCE provides a mapping interface between the UNIX System Services Threads and DCE Threads. Chapter 19, “Migrating between UNIX System Services and DCE Threads” on page 361 shows you how to invoke either the DCE Threads package or the UNIX System Services Threads package for your applications. For compatibility with other DCE implementations your applications need to invoke the DCE Threads package. Chapter 19, “Migrating between UNIX System Services and DCE Threads” on page 361 also describes the differences between DCE Threads and UNIX System Services Threads so you can migrate between the two packages if you require.

**Note:** The following threads function that may be available in the Distributed Computing Environment product from OSF, or on DCE offerings from other vendor, are not supported in z/OS DCE:

- The following interfaces are not supported and return -1, errno ENOSYS:
  - pthread_attr_getinheritsched()
  - pthread_attr_getprio()
  - pthread_attr_getsched()
  - pthread_attr_setinheritsched()
  - pthread_attr_setprio()
  - pthread_attr_setsched()
  - pthread_getprio()
  - pthread_getscheduler()
  - pthread_setprio()
  - pthread_setscheduler()
- For all pthread interfaces (including mutexes, threads, condition variables and so on), the interfaces do not accept copies of the objects as a parameter. The object returned from the pthread interface to create the object must be used at all times.
- Unlike the OSF DCE implementation, the z/OS DCE implementation of the following functions can raise an exception (**exc_e_cpa_error**) in error situations:
  - pthread_lock_global_np()
  - pthread_unlock_global_np()
- pthread_cond_timedwait() expects an absolute hardware time (that is, time-of-day clock value) for the wait time instead of the DCE software clock time, which is what OSF/DCE expects. pthread_get_expiration_np() returns a software adjusted time as in the OSF/DCE model, and is used as input to pthread_cond_timedwait().
- exc_report() does not print out a message to stderr as expected. z/OS DCE uses Reliability, Availability and Serviceability (RAS) services to log messages instead of this function.
- **pthread_mutex_init** cannot initialize a mutex more than once.

# Chapter 13. Introduction to Multithreaded Programming

DCE Threads is a user-level (non-kernel) threads package based on the pthreads interface specified by POSIX in 1003.4a, Draft 4. This chapter introduces multithreaded programming, which is the division of a program into multiple threads (parts) that execute concurrently. In addition, this chapter describes four software models that improve multithreaded programming performance.

A ***thread*** is a single sequential flow of control within a program. It is the active execution of a designated routine, including any nested routine invocations. Within a single thread, there is a single point of execution. Most traditional programs consist of a single thread.

Each thread has its own thread identifier, scheduling policy and priority, *errno* value, thread-specific data bindings, and the required system resources to support a flow of control.

## Advantages of Using Threads

With a thread package, a programmer can create several threads within a process. Threads execute concurrently, and within a multithreaded process, there are multiple points of execution at any time. Threads execute within a single address space. Multithreaded programming offers the following advantages:

- Performance

  Threads improve the performance (throughput, computational speed, responsiveness, or some combination) of a program. Multiple threads are useful in a multiprocessor system where threads run concurrently on separate processors. In addition, multiple threads also improve program performance on single-processor systems by permitting the overlap of input and output or other slow operations with computational operations.

  Think of threads as running simultaneously, regardless of the number of processors present. You cannot make any assumptions about the start or finish times of threads or the sequence in which they execute, unless they are explicitly synchronized.

- Shared Resources

  When you use multiple threads instead of separate processes multiple threads share a single address space, all open files, and other resources.

- Potential Simplicity

  Multiple threads may reduce the complexity of some applications that are inherently suited for threads.

## Software Models for Multithreaded Programming

The following sections describe four software models for which multithreaded programming is especially well suited:

- Boss/worker model
- Work crew model
- Pipelining model
- Combinations of models

## Boss/Worker Model

In a boss/worker model of program design, one thread functions as the boss because it assigns tasks to worker threads. Each worker performs a different type of task until it is finished, at which point the worker interrupts the boss to indicate that it is ready to receive another task. Alternatively, the boss polls workers periodically to see whether or not each worker is ready to receive another task.

A variation of the boss/worker model is the work queue model. The boss places tasks in a queue, and workers check the queue and take tasks to perform. An example in an office environment is a secretarial typing pool. The office manager puts documents to be typed in a basket, and typists take documents from the basket to work on.

## Work Crew Model

In the work crew model, multiple threads work together on a single task. The task is divided into pieces that are performed in parallel, and each thread performs one piece. An example is a group of people cleaning a house. Each person cleans certain rooms or performs certain types of work (washing floors, polishing furniture, and so forth), and each works independently.

The following figure shows a task performed by three threads in a work crew model.

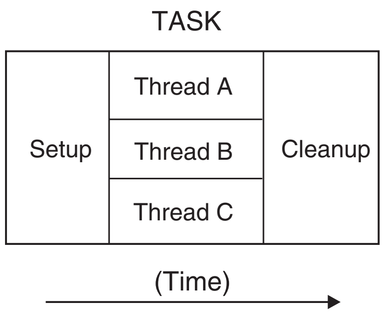


*Figure 55. Work Crew Model*

## Pipelining Model

In the pipelining model, a task is divided into steps. The steps must be performed in sequence to produce a single instance of the desired output. The work done in each step (except for the first and last) is based on the preceding step and is a prerequisite for the work in the next step. However, the program is designed to produce multiple instances of the desired output, and the steps are designed to operate in a parallel time frame so that each step is kept busy.

An example is an automobile assembly line. Each step or stage in the assembly line is continually busy receiving the product of the previous stage's work, performing its assigned work, and passing the product along to the next stage. A car needs a body before it can be painted, but at any one time numerous cars are receiving bodies, and then they are being painted.

In a multithreaded program using the pipelining model, each thread represents a step in the task. The following figure shows a task performed by three threads in a pipelining model.

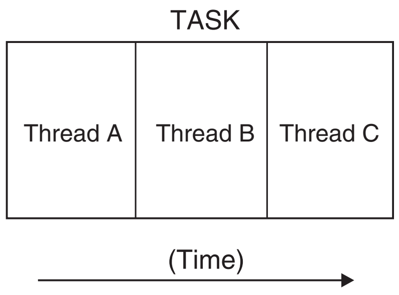


*Figure 56. Pipelining Model*

## Combinations of Models

You may find it appropriate to combine the software models in a single program if your task is complex. For example, a program could be designed using the pipelining model, but one or more steps could be handled by a work crew. In addition, tasks could be assigned to a work crew by taking a task from a work queue and deciding (based on the task characteristics) which threads are needed for the work crew.

## Potential Disadvantages with Multithreaded Programming

When you design and code a multithreaded program, consider the following problems and accommodate or eliminate each problem as appropriate:

- Potential Complexity

  The level of expertise required for designing, coding, and maintaining multithreaded programs may be higher than for most single-threaded programs because multithreaded programs might need shared access to resources, mutexes, and condition variables. Weigh the potential benefits against the complexity and its associated risks.

- Nonreentrant Software

  If a thread calls a routine or library that is not reentrant, use the global locking mechanism to prevent the nonreentrant routines from modifying a variable that another thread modifies. "Avoiding Nonreentrant Software" on page 334 discusses nonreentrant software in more detail.

  **Note:** A multithreaded program must be reentrant; that is, it must allow multiple threads to execute at the same time. Therefore, be sure that your compiler generates reentrant code before you do any design or coding work for multithreading.

  If your program is nonreentrant, any thread synchronization techniques that you use are not guaranteed to be effective.

- Race Conditions

  A race condition is a type of programming error which causes unpredictable and erroneous program behavior. "Race Conditions" on page 335 discusses race conditions in more detail.

- Deadlocks

  A type of programming error called a deadlock causes two or more threads to be blocked from executing. "Deadlocks" on page 336 discusses deadlocks in more detail.

- Blocking Calls

  Certain system or library calls may cause an entire process to block while waiting for the call to complete. As a result, all other threads stop executing.

# Chapter 14. Thread Concepts and Operations

This chapter discusses concepts and techniques related to DCE Threads.

For detailed information on the multithreading routines referred to in this chapter, see the *z/OS DCE Application Development Reference*.

## Thread Operations

A thread changes states as it runs, waits to synchronize, or is ready to be run. A thread is in one of the following states:

| | |
|---|---|
| Waiting | The thread is not eligible to execute because it is synchronizing with another thread or with an external event. |
| Ready | The thread is eligible to be executed by a processor. |
| Running | The thread is currently being executed by a processor. |
| Terminated | The thread has completed all of its work. |

Figure 57 shows the transitions between states for a typical thread implementation.

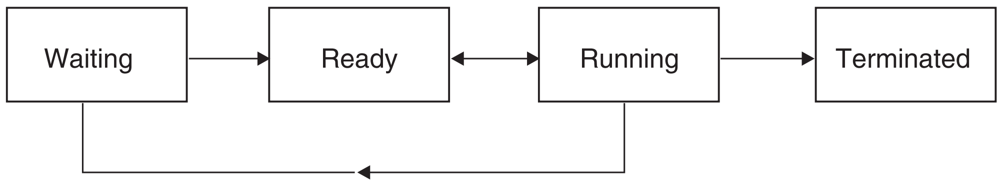


*Figure 57. Thread State Transitions*

The operations that you can perform include:

- Starting
- Waiting for
- Terminating
- Deleting threads.

## Starting a Thread

To start a thread, create it using the **pthread_create()** routine. It creates the thread, assigns specified or default attributes, and starts execution of the function you specified as the thread's start routine. A unique identifier (handle) for that thread is returned from the **pthread_create()** routine.

## Terminating a Thread

A thread exists until it terminates and the **pthread_detach()** routine is called for the thread. The **pthread_detach()** routine can be called for a thread before or after it terminates. If the thread terminates before **pthread_detach()** is called for, the thread continues to exist and can be synchronized (joined) until it is detached. Thus, the object (thread) can be detached by any thread that has access to a handle to the object.

**Note:** **pthread_detach()** must be called to release the memory allocated for the thread objects so that this storage does not build up and cause the process to run out of memory. For example, after a thread returns from a call to join, it detaches the joined-to thread if no other threads join with it. Similarly, if a thread has no other threads joining with it, it detaches itself so that its thread object is deallocated as soon as it terminates.

A thread terminates for any of the following reasons:

- The thread returns from its start routine; this is the usual case.
- The thread calls the **pthread_exit()** routine.

  The **pthread_exit()** routine terminates the calling thread and returns a status value indicating the thread's exit status to any potential joiners.
- The thread is canceled by a call to the **pthread_cancel()** routine.

  The **pthread_cancel()** routine requests termination of a specified thread if cancelation is permitted. See “Thread Cancelation” on page 328 for more information on canceling threads and controlling whether cancelation is permitted.
- An error occurs in the thread.

  Examples of errors that cause thread termination are programming errors, segmentation faults, or exceptions that are not handled.

## Waiting for a Thread to Terminate

A thread waits for the termination of another thread by calling the **pthread_join()** routine. Execution in the current thread is suspended until the specified thread terminates. If multiple threads call this routine and specify the same thread, all threads resume execution when the specified thread terminates.

Do not confuse **pthread_join()** with other routines that cause waits, and that are related to the use of a particular multithreading feature. For example, use **pthread_cond_wait()** or **pthread_cond_timedwait()** to wait for a condition variable to be signaled or broadcast. (See “Condition Variables” on page 325 for information about condition variables.)

## Deleting a Thread

A thread is automatically deleted after it terminates; that is, no explicit deletion operation is required. Use **pthread_detach()** to free the storage of a terminated thread. Use **pthread_cancel()** to request that a running thread terminate itself.

If the thread has not yet terminated, the **pthread_detach()** routine marks the thread for deletion, and its storage is reclaimed immediately when the thread terminates. A thread cannot be joined or canceled after the **pthread_detach()** routine is called for the thread, even if the thread has not yet terminated.

If a thread that is not detached terminates, its storage remains so that other threads can join with it. Storage is reclaimed when the thread is eventually detached.

For more information, see “Terminating a Thread” on page 319.

## New Primitives

Routines implemented by DCE Threads that are not specified by Draft 4 of the POSIX 1003.4a standard are indicated by an **_np** suffix on the name. These routines have not been incorporated into the POSIX standard, and as such are extensions to that document.

## Attributes Objects

An attributes object is used to describe the behavior of threads, mutexes, and condition variables. This description consists of the individual attribute values that are used to create an attributes object. Whether an attribute is valid depends on whether it describes threads, mutexes, or condition variables.

When you create an object, you can accept the default attributes for that object, or you can specify an attributes object that contains individual attributes that you have set.

The following subsections describe how to create and delete attributes objects and describe the individual attributes that you can specify for different objects.

### Creating an Attributes Object

To create an attributes object, use one of the following routines, depending on the type of object to which the attributes apply:

| | |
|---|---|
| **pthread_attr_create()** | For thread attributes objects |
| **pthread_condattr_create()** | For condition variable attributes objects |
| **pthread_mutexattr_create()** | For mutex attributes objects. |

These routines create an attributes object containing default values for the individual attributes. To modify any attribute values in an attributes object, use one of the set routines described in the following subsections.

Creating an attributes object or changing the values in an attributes object does not affect the attributes of objects previously created.

### Deleting an Attributes Object

To delete an attributes object, use one of the following routines:

| | |
|---|---|
| **pthread_attr_delete()** | For thread attributes objects |
| **pthread_condattr_delete()** | For condition variable attributes objects |
| **pthread_mutexattr_delete()** | For mutex attributes objects. |

Deleting an attributes object does not affect the attributes of objects previously created.

## Thread Attributes

**Note to Readers**

The DCE Threads stacksize attribute below is the only attribute supported by z/OS DCE.

With a thread attributes object you can specify values for thread attributes other than the defaults when you create a thread with the **pthread_create()** routine. To use a thread attributes object, perform the following steps:

1. Create a thread attributes object by calling the **pthread_attr_create()** routine.
2. Call the routines discussed in the following subsections to set the individual attributes of the thread attributes object.
3. Create a new thread by calling the **pthread_create()** routine and specifying the identifier of the thread attributes object.

With z/OS DCE, you can only change the stacksize thread attribute for a new thread. A new thread is assigned the default values for the remaining thread attributes listed below. You cannot change the following thread attributes on z/OS DCE as you might on other DCE platforms:

- Scheduling policy attribute
- Scheduling priority attribute
- Inherit scheduling attribute

The UNIX System Services Threads facility supports other threads objects, such as thread weight, detachstate, as well as stacksize. Note that for DCE applications, threads are always MEDIUM weight by default. Refer to *z/OS UNIX System Services Programming Tools* for information on how to set these attributes.

**Scheduling Policy Attribute:** The scheduling policy attribute describes the overall scheduling policy of the threads in your application.

z/OS DCE does not support any control over scheduling policy. Threads are dispatched as first in first out (FIFO) by default.

"Thread Scheduling" on page 329 describes and shows the effect of scheduling policy on thread scheduling.

**Scheduling Priority Attribute:** With z/OS DCE, all threads in a process have equal priority. There is no preferential treatment of any threads in terms of mutexes or condition variables.

**Inherit Scheduling Attribute:** On z/OS DCE, a newly created thread inherits the scheduling attributes (scheduling priority and policy) of the creating thread by default.

**Stacksize Attribute:** The stacksize attribute is the minimum size (in bytes) of the memory required for a thread's stack. The default value for z/OS DCE is 512 kilobytes.

You can change the stack size from the default by setting the STACK Language Environment runtime option. You can also use the **#pragma** directive to set this option in your application code. See the *z/OS DCE Application Development Guide: Introduction and Style* for information on recommended Language Environment runtime options to use with z/OS DCE applications.

## Mutex Attributes

Use a mutex attributes object to specify values for mutex attributes (other than the defaults) when you create a mutex with the **pthread_mutex_init()** routine.

The mutex type attribute specifies whether a mutex is fast, recursive, or nonrecursive. (See “Mutexes” for definitions.) Set the mutex type attribute by calling the **pthread_mutexattr_setkind_np()** routine. (Any routine with the **_np** suffix is nonportable). If you do not use a mutex attributes object to select a mutex type, calling the **pthread_mutex_init()** routine creates a fast mutex by default.

## Condition Variable Attributes

Currently, attributes affecting condition variables are not defined. You cannot change any attributes in the condition variable attributes object.

“Condition Variables” on page 325 describes the purpose and uses of condition variables.

## Synchronization Objects

In a multithreaded program, you must use synchronization objects whenever there is a possibility of corruption of shared data or conflicting scheduling of threads that have mutual scheduling dependencies. The following subsections discuss two kinds of synchronization objects: mutexes and condition variables.

## Mutexes

A ***mutex*** (**mut**ual **ex**clusion) is an object that multiple threads use to ensure the integrity of a shared resource that they access, most commonly shared data. A mutex has two states: locked and unlocked. For each piece of shared data, all threads accessing that data must use the same mutex. Each thread locks the mutex before it accesses the shared data and unlocks the mutex when it is finished accessing that data. If the mutex is locked by another thread, the thread requesting the lock is blocked when it tries to lock the mutex if you call **pthread_mutex_lock()**. The blocked thread continues and is not blocked if you call **pthread_mutex_trylock()**.

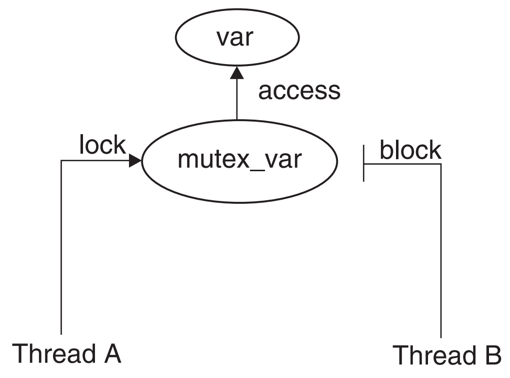


*Figure 58. Only One Thread Can Lock a Mutex*

Each mutex must be initialized. (To initialize mutexes as part of the program’s one-time initialization code, see “One-Time Initialization Routines” on page 327.) To initialize a mutex, use the **pthread_mutex_init()** routine. With this routine, you specify an attributes object, which allows you to specify the mutex type. The following are types of mutexes:

- A fast mutex (the default) is locked only once by a thread. If the thread tries to lock the mutex again without first unlocking it, the thread waits for itself to release the first lock and deadlocks on itself.

  **Note:** With z/OS DCE, a fast mutex is the same as a nonrecursive mutex. That is, although the **FAST_MUTEX** attribute exists, it adopts the behavior of a nonrecursive mutex, which is described below.

- A recursive mutex can be locked more than once by a given thread without causing a deadlock. The thread must call the **pthread_mutex_unlock()** routine the same number of times that it called the **pthread_mutex_lock()** routine before another thread can lock the mutex.

  Recursive mutexes have the notion of a mutex owner. When a thread successfully locks a recursive mutex, it owns that mutex and the lock count is set to 1. Any other thread attempting to lock the mutex blocks until the mutex becomes unlocked. If the owner of the mutex attempts to lock the mutex again, the lock count increments, and the thread continues running. When an owner unlocks a recursive mutex, the lock count decrements. The mutex remains locked and owned until the count reaches 0 (zero). It is an error for any thread other than the owner to attempt to unlock the mutex.

  A recursive mutex is useful if a thread needs exclusive access to a piece of data, and it needs to call another routine (or itself) that needs exclusive access to the data. A recursive mutex allows nested attempts to lock the mutex to succeed rather than deadlock.

  This type of mutex requires more careful programming. Never use a recursive mutex with condition variables because the implicit unlock performed for a **pthread_cond_wait()** or **pthread_cond_timedwait()** may not actually release the mutex. In that case, no other thread can satisfy the condition of the predicate.

- A nonrecursive mutex is locked only once by a thread, like a fast mutex. If the thread tries to lock the mutex again without first unlocking it, the thread receives an error. Thus, nonrecursive mutexes are more informative than fast mutexes because fast mutexes block in such a case, leaving it up to you to determine the reasons thread no longer executes. If someone other than the owner tries to unlock a nonrecursive mutex, an error is returned.

To lock a mutex, use one of the following routines, depending on what you want to happen if the mutex is locked:

| | |
|---|---|
| **pthread_mutex_lock()** | If the mutex is locked, the thread waits for the mutex to become available. |
| **pthread_mutex_trylock()** | If the mutex is locked, the thread continues without waiting for the mutex to become available. The thread immediately checks the return status to see if the lock was successful, and then takes whatever action is appropriate if it was not. |

When a thread is finished accessing a piece of shared data, it unlocks the associated mutex by calling the **pthread_mutex_unlock()** routine.

If another thread is waiting on the mutex, its execution is unblocked. If more than one thread is waiting on the mutex, the scheduling policy and the thread scheduling priority determine which thread acquires the mutex.

You can delete a mutex and reclaim its storage by calling the **pthread_mutex_destroy()** routine. Use this routine only after the mutex is no longer needed by any thread. Mutexes are automatically deleted when the program terminates.

## Condition Variables

A ***condition variable*** allows a thread to block its own execution until some shared data reaches a particular state. Cooperating threads check the shared data and wait on the condition variable. For example, one thread in a program produces work-to-do packets and another thread consumes these packets (does the work). If the work queue is empty when the consumer thread checks it, that thread waits on a work-to-do condition variable. When the producer thread puts a packet on the queue, it signals the work-to-do condition variable.

A condition variable is used to wait for a shared resource to assume some specific state (a predicate). A mutex, on the other hand, is used to protect some shared resource while the resource is being manipulated.

For example, a thread A may need to wait for a thread B to finish a task X before thread A proceeds to execute a task Y. Thread B can tell thread A that it has finished task X by using a variable to which they both have access, a condition variable. When thread A is ready to execute task Y, it looks at the condition variable to see if thread B is finished. (Figure 59 illustrates this.)

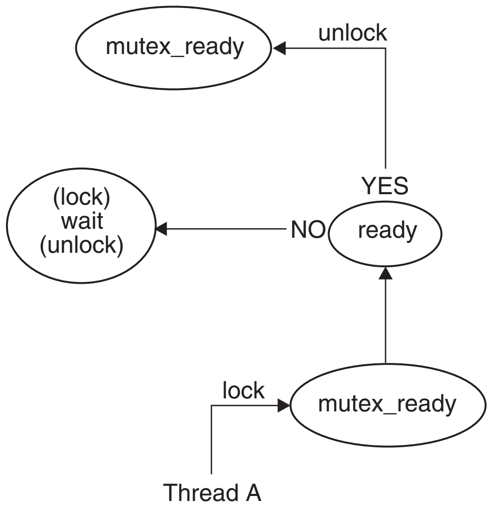


*Figure 59. Thread A Waits on Condition Ready, Then Wakes Up and Proceeds*

First, thread A locks the mutex named **mutex_ready** that is associated with the condition variable. Then it reads the predicate associated with the condition variable named **ready**. If the predicate indicates that thread B has finished task X, then thread A can unlock the mutex and proceed with task Y. If the condition variable predicate indicates that thread B has not yet finished task X, thread A waits for the condition variable to change. Thread A calls the **wait** primitive. Waiting on the condition variable automatically unlocks the mutex, allowing thread B to lock the mutex when it has finished task X as shown in Figure 60 on page 326.

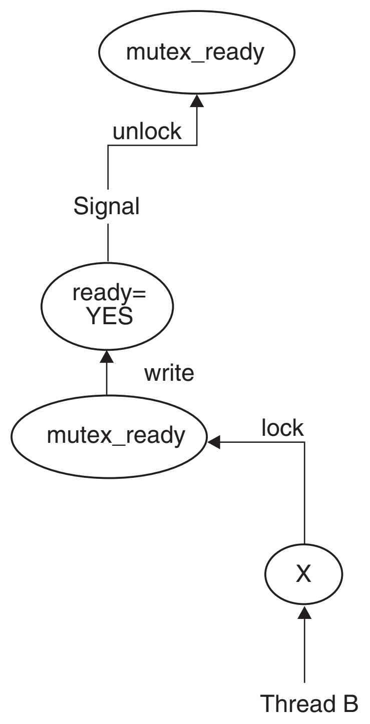


*Figure 60. Thread B Signals Condition Ready*

Thread B updates the predicate named **ready** associated with the condition variable to the state thread A is waiting for. It also executes a signal on the condition variable while holding the mutex **mutex_ready**. Thread A wakes up, verifies that the condition variable is in the correct state, and proceeds to execute task Y as shown in Figure 59 on page 325.

**Note:** Although the condition variable is used for explicit communication among threads, the communication is anonymous. Thread B does not necessarily know that thread A is waiting on the condition variable that thread B signals. And thread A does not know that it was thread B that woke it up from its wait on the condition variable.

Use the **pthread_cond_init()** routine to create a condition variable. To create condition variables as part of the program's one-time initialization code, see "One-Time Initialization Routines" on page 327.

Use the **pthread_cond_wait()** routine to cause a thread to wait until the condition is signaled or broadcast. This routine specifies a condition variable and a mutex that you have locked. (If you have not locked the mutex, the results of **pthread_cond_wait()** are unpredictable.) This routine unlocks the mutex and causes the calling thread to wait on the condition variable until another thread calls one of the following routines:

| | |
|---|---|
| **pthread_cond_signal()** | To wake one thread that is waiting on the condition variable |
| **pthread_cond_broadcast()** | To wake all threads that are waiting on a condition variable. |

If you want to limit the time that a thread waits for a condition to be signaled or broadcast, use the **pthread_cond_timedwait()** routine. This routine specifies the condition variable, mutex, and absolute time when the wait should expire, if the condition variable is not signaled or broadcast.

You can delete a condition variable and reclaim its storage by calling the **pthread_cond_destroy()** routine. Use this routine only after the condition variable is no longer needed by any thread. Condition variables are automatically deleted when the program terminates.

## Other Synchronization Methods

There is another synchronization method that is not anonymous — the **join** primitive. With this primitive, one thread can wait for another specific thread to complete its execution. When the second thread is finished, the first thread unblocks and continues its execution. Unlike mutexes and condition variables, the join primitive is not associated with any particular shared data.

## One-Time Initialization Routines

You probably have one or more routines that must be executed before any thread executes code in your application, but must be executed only once regardless of the sequence in which threads start executing. For example, you may want to create mutexes and condition variables (each of which must be created only once) in an initialization routine. Multiple threads can call the **pthread_once()** routine, or the **pthread_once()** routine can be called multiple times in the same thread, resulting in only one call to the specified routine.

Use the **pthread_once()** routine to ensure that your application initialization routine is executed only a single time, that is, by the first thread that tries to initialize the application. This routine is the only way to guarantee that one-time initialization is performed in a multithreaded environment on a given platform. The **pthread_once()** routine is of particular use for runtime libraries, which are often called the first time after multiple threads are created.

## Thread-Specific Data

The thread-specific data interfaces allow each thread to associate an arbitrary value with a shared key value created by the program.

Thread-specific data is like a global variable in which each thread can keep its own value, but is accessible to the thread anywhere in the program.

Use the following routines to create and access thread-specific data:

| | |
|---|---|
| **pthread_keycreate()** | To create a unique key value. |
| **pthread_setspecific()** | To associate data with a key. |
| **pthread_getspecific()** | To obtain the data associated with a key. |

The **pthread_keycreate()** routine generates a unique key value that is shared by all threads in the process. This key is the identifier of a piece of thread-specific data. Each thread uses the same key value to assign or retrieve a thread-specific value. This keeps your data separate from other thread-specific data. One call to the **pthread_keycreate()** routine creates a cell in all threads. Call this routine to specify a routine to be called to destroy the context value associated with this key when the thread terminates.

The **pthread_setspecific()** routine associates the address of some data with a specific key. Multiple threads associate different data (by specifying different addresses) with the same key. For example, each thread points to a different block of dynamically allocated memory that it has reserved.

The **pthread_getspecific()** routine obtains the address of the thread-specific data value associated with a specified key. Use this routine to locate the data associated with the current thread's context.

## Thread Cancelation

***Canceling*** is a mechanism by which one thread terminates another thread (or itself). When you request that a thread be canceled, you are requesting that it terminate as soon as possible. However, the target thread can control how quickly it terminates by controlling its general cancelability and its asynchronous cancelability.

The following is a list of the pthread routines that are cancelation points:

- **pthread_setasynccancel()**
- **pthread_testcancel()**
- **pthread_delay_np()**
- **pthread_join()**
- **pthread_cond_wait()**
- **pthread_cond_timedwait()**.

General cancelability (referred to as controlled cancelability by UNIX System Services Threads) is enabled by default. A thread is canceled only at specific places in the program, for example, when the **pthread_cond_wait()** routine is called. If general cancelability is enabled, request the delivery of any pending cancel request by using the **pthread_testcancel()** routine. With this routine, you can permit cancelation to occur at places where it may not otherwise be permitted under general cancelability. It is especially useful within very long loops to ensure that cancel requests are noticed within a reasonable time.

If you disable general cancelability, the thread cannot be terminated by any cancel request. Disabling general cancelability means that a thread could wait indefinitely if it does not come to a normal conclusion; therefore, be careful about disabling general cancelability.

Asynchronous cancelability, when enabled, allows cancels to be delivered to the enabling thread at any time, not only at those times that are permitted when just general cancelability is enabled. Use asynchronous cancelation primarily during long processes that do not have specific places for cancel requests. Asynchronous cancelability is disabled by default. Disable asynchronous cancelability when calling threads routines or any other runtime library routines that are not explicitly documented as cancel-safe.

**Note:** If general cancelability is disabled, the thread cannot be canceled, regardless of whether asynchronous cancelability is enabled or disabled. The setting of asynchronous cancelability is relevant only when general cancelability is enabled.

Use the following routines to control the canceling of threads:

| | |
|---|---|
| **pthread_setcancel()** | To enable and disable general cancelability |
| **pthread_testcancel()** | To request delivery of a pending cancel to the current thread |
| **pthread_setasynccancel()** | To enable and disable asynchronous cancelability |
| **pthread_cancel()** | To request that a thread be canceled. |

## Thread Scheduling

**Note to Readers**

Thread scheduling is not supported by z/OS DCE. The default thread scheduling policy is first in first out.

# Chapter 15. Programming with Threads

This chapter discusses issues you face when writing a multithreaded program and how to deal with those issues.

The following topics are discussed in this chapter:

- Using signals
- Avoiding nonreentrant software
- Using synchronization objects.

**Note to Readers**

z/OS DCE does not support the **fork()** system call for DCE applications. As a consequence, the **atfork()** system call is also unsupported. As system calls using z/OS DCE are thread-reentrant, the **fork()** and **atfork()** jacket routines are also not supported on z/OS DCE. If you transfer your DCE application from another platform that supports **fork()** and **atfork()**, you have to remove these calls to run your application using z/OS DCE.

## Using Signals

The following subsections cover three topics: types of signals, DCE Threads signal handling, and alternatives to using signals.

## Types of Signals

Signals are delivered as a result of some event. Such signals are grouped into the following four categories of pairs that are orthogonal to each other:

- Terminating and synchronous
- Terminating  and asynchronous
- Nonterminating and synchronous
- Nonterminating and asynchronous.

The action that DCE Threads takes when a particular signal is delivered depends on the characteristics of that signal.

**Terminating Signals:** Terminating signals result in the termination of the process by default. Whether a particular signal is terminating or not is independent of whether it is synchronously or asynchronously delivered.

**Nonterminating Signals:** Nonterminating signals do not result in the termination of the process by default.

Nonterminating signals represent events that can be either internal or external to the process. The process may require notification about or ignore these events. When a nonterminating asynchronous signal is delivered to the process, DCE Threads awakens any threads that are waiting for the signal. This is the only action that DCE Threads takes, because the signal has no effect by default.

**Synchronous Signals:** Synchronous signals are the result of an event that occurs inside a process and are delivered synchronously with respect to that event. For example, if a floating-point calculation results in an overflow, then a **SIGFPE** (floating-point exception signal) is delivered to the process immediately following the instruction that resulted in the overflow.

The default behavior for DCE Threads when a synchronous terminating signal occurs, is not to handle the signal, possibly resulting in a core dump. If an application using DCE Threads wants to handle signals, it must set up a signal handler by calling **sigaction()**. Note that the DCE Threads behavior is similar to the default behavior of most UNIX programs.

Synchronous, terminating signals represent an error that has occurred in the currently executing thread.

**Asynchronous Signals:** Asynchronous signals are the result of an event that is external to the process and are delivered at any point in a thread's execution when such an event occurs. For example, when a user running a program enters **kill SIGINT process** (running under the UNIX Emulator), a **SIGINT** (interrupt signal) is delivered to the process.

Asynchronous, terminating signals represent an occurrence of an event that is external to the process, and if it is not handled, results in the termination of the process. When an asynchronous terminating signal is delivered, DCE Threads catches it and checks to see if any threads are waiting for it. If threads are waiting, they are awakened, and the signal is considered handled and is dismissed. If there are no waiting threads, then DCE Threads causes the process to be terminated as if the signal had not been handled.

## DCE Threads Signal Handling

DCE Threads provides the POSIX **sigwait()** service to allow threads to perform activities similar to signal handling without having to deal with signals directly. It also provides a jacket for **sigaction()** that allows each thread to have its own handler for synchronous signals.

To provide these mechanisms, DCE Threads installs signal handlers for most of the signals during initialization.

DCE Threads do not provide handlers for several signals. These signals and a reason why handlers are not provided are listed below.

| Signal | Reason Handler Is Not Provided |
|---|---|
| **SIGKILL** and **SIGSTOP** | These signals cannot be caught by user mode code. |
| **SIGTRAP** | Catching this signal interferes with debugging. |
| **SIGTSTP** and **SIGQUIT** | These signals are caught only while a thread has issued a **sigwait()** call, because their default actions are otherwise valuable. |

**The POSIX sigwait() Service:** The DCE Threads implementation of the POSIX **sigwait()** service allows any thread to block until one of a specified set of signals is delivered. A thread waits for any of the asynchronous signals, except for **SIGKILL** and **SIGSTOP**.

A thread cannot wait for a synchronous signal. Synchronous signals are the result of an error during the execution of a thread, and if the thread is waiting for a signal, then it is not executing. Therefore, a synchronous signal cannot occur for a particular thread while it is waiting, and so the thread waits forever.

**The POSIX sigaction() Service:** The DCE Threads implementation of the POSIX **sigaction()** service allows for per-thread handlers to be installed for catching synchronous signals. The **sigaction()** routine only modifies behavior for individual threads and only works for synchronous signals. Setting the signal action to **SIG_DFL** for a specific signal will restore the thread's default behavior for that signal. Attempting to set a signal action for an asynchronous signal is an error.

### Alternatives to Using Signals

Avoid using signals in multithreaded programs. DCE Threads provides alternatives to signal handling. These alternatives are discussed in "Using Synchronization Objects" on page 335 and "Signaling a Condition Variable" on page 336.

**Note:** In order to implement these alternatives, DCE Threads must install its own signal handlers. These are installed when DCE Threads initializes itself.

Following are two reasons for avoiding signals:

- Signals cannot be used in a modular way in a multithreaded program.
- Signals, used as an asynchronous programming technique, are unnecessary in a multithreaded program.

In a multithreaded program, signals cannot be used in a modular way because, as on most current implementations of UNIX, signals are inherently a process construct. There is only one instance of each signal and of each signal handler routine for all of the threads in an application. If one thread handles a particular signal in one way, and a different thread handles the same signal in a different way, then the thread that installs its signal handler last handles the signal. This rule applies only to asynchronously generated signals; synchronous signals are handled on a per-thread basis when using DCE Threads, that is, when **_DCE_THREADS** is defined.

Do not use asynchronous programming techniques in conjunction with threads, particularly those that increase parallelism, such as using timer signals and I/O signals. These techniques can be complicated. They are also unnecessary because threads provide a mechanism for parallel execution that is simpler and less prone to error where concurrence can be of value. Furthermore, most of the threads routines are not supported for use in interrupt routines (such as signal handlers), and portions of runtime libraries cannot be used reliably inside a signal handler.

## Nonthreaded Libraries

As programming with threads becomes common practice, you need to ensure that threaded code and nonthreaded code (code that is not designed to work with threads) work properly together in the same application. For example, you may write a new application that uses threads (for example, an RPC server) and link it with a library that does not use threads (and is thus not thread-safe). In such a situation, you can do one of the following:

- Work with the nonthreaded software

- Change the nonthreaded software to be thread-safe.

## Working with Nonthreaded Software

Thread-safe code is code that works properly in a threaded environment. To work with nonthread-safe code, associate the global lock with all calls to such code.

You can implement the lock on the side of the routine user or the routine provider. For example, you can implement the lock on the side of the routine user if you write a new application like an RPC server that uses threads, and you link it with a library that does not. Or, if you have access to the nonthreaded code, the locks can be placed on the side of the routine provider, within the actual routine. Implement the locks as follows:

1. Associate one lock, a global lock, with execution of such code.
2. Require all of your threads to lock prior to execution of nonthreaded code.
3. Perform an unlock when execution is complete.

By using the global lock, you ensure that only one thread executes in outside libraries that may call each other, and in unknown code. Using a single global lock is safer than using multiple local locks because it is difficult to be aware of everything a library may be doing or of the interactions that library can have with other libraries.

### Changing Nonthreaded Code to Be Thread-Reentrant

Thread-reentrant code is code that works properly while multiple threads execute it concurrently. Thread-reentrant code is thread-safe, but thread-safe code may not be thread-reentrant. Document your code as being thread-safe or thread-reentrant.

More work is involved in making code thread-reentrant than in making code thread-safe. To make code thread-reentrant, do the following:

- Use proper locking protocols to access global or static variables.
- Use proper locking protocols when you use code that is not thread-safe.
- Store thread-specific data on the stack or heap.
- Ensure that the compiler produces thread-reentrant code.
- Document your code as being thread-reentrant.

## Avoiding Nonreentrant Software

The following subsections discuss two methods to help you avoid the pitfalls of nonreentrant software. These methods are:

- Global lock
- Thread-specific storage.

## Global Lock

Use a global lock that has the characteristics of a recursive mutex, instead of a regular mutex when calling routines that you think are nonreentrant. (When in doubt, assume the code is nonreentrant.)

The **pthread_lock_global_np()** routine is a locking protocol that is used to call nonreentrant routines, often found in existing library packages that were not designed to run in a multithreaded environment.

The way to call a library function that is not reentrant from a multithreaded program is to protect the function with a mutex. If every function that calls a library locks a particular mutex before the call and releases the mutex after the call, then the function completes without interference. However, this is difficult to do successfully because the function may be called by many libraries. A global lock solves this problem by providing a universal lock. Any code that calls any nonreentrant function uses the same lock.

To lock a global lock, call the **pthread_lock_global_np**() routine. To unlock a global lock, call the **pthread_unlock_global_np**() routine.

## Thread-Specific Storage

To avoid nonreentrancy when writing new software, avoid using global variables to store data that is thread-specific data. See “Thread-Specific Data” on page 327 for more information.

Alternatively, allocate thread-specific data on the stack or heap and explicitly pass its address to called routines.

## Using Synchronization Objects

The following subsections discuss the use of mutexes to prevent two potential problems: race conditions and deadlocks. Also discussed are reasons why you should signal a condition variable with the associated mutex locked.

## Race Conditions

A *race condition* occurs when two or more threads perform an operation, and the result of the operation depends on unpredictable timing factors; specifically, when each thread executes and waits and when each thread completes the operation.

An example of a race condition:

1. Both A and B are executing (X = X + 1).
2. A reads the value of X (for example, X = 5).
3. B then reads the value of X and increments it (making X = 6).
4. A is rescheduled and now increments X. Based on its earlier read operation, A reads X as (X+1 = 5+1 = 6). X is now 6. X should be 7 because it was incremented once by A and once by B.

To avoid race conditions, ensure that any variable modified by more than one thread has only one mutex associated with it. Do not assume that a simple add operation can be completed without allowing another thread to execute. Such operations are generally not portable, especially to multiprocessor systems. If it is possible for two threads to share a data point, use a mutex.

## Deadlocks

A *deadlock* occurs when one or more threads are permanently blocked from executing because each thread waits on a resource held by another thread in the deadlock. A thread can also deadlock on itself.

The following is one technique for avoiding deadlocks:

1. Associate a sequence number with each mutex.
2. Lock mutexes in sequence.
3. Do not attempt to lock a mutex with a sequence number lower than that of a mutex the thread already holds.

Another technique — useful when a thread needs to lock the same mutex more than once before unlocking it — is to use a recursive mutex. This technique prevents a thread from deadlocking on itself.

## Signaling a Condition Variable

When you are signaling a condition variable and that signal may cause the condition variable to be deleted, you should signal or broadcast with the mutex locked.

The recommended coding for signaling a condition variable appears at the end of this chapter. The following two C code fragments show coding that is not recommended. The following C code fragment is executed by a releasing thread:

```
pthread_mutex_lock (m);
.../* Change shared variables to allow another thread to proceed */
pthread_mutex_unlock (m);   <---- Point A
pthread_cond_signal (cv);   <---- Statement 1
```

The following C code fragment is executed by a potentially blocking thread:

```
pthread_mutex_lock (m);
while (!predicate ...
    pthread_cond_wait (cv, m);

pthread_mutex_unlock (m);
```

**Note:** It is possible for a potentially blocking thread to be running at Point A while another thread is interrupted. The potentially blocking thread can then see the predicate true, and therefore not become blocked on the condition variable.

Signaling a condition variable without first locking a mutex is not a problem. However, if the released thread deletes the condition variable without any further synchronization at Point A, then the releasing thread will fail when it attempts to execute Statement 1 because the condition variable no longer exists.

This problem occurs when the releasing thread is a worker thread and the waiting thread is the boss thread, and the last worker thread tells the boss thread to delete the variables that are being shared by boss and worker.

The following C code fragment shows the recommended coding for signaling a condition variable while the mutex is locked:

```
pthread_mutex_lock (m);
.../* Change shared variables to allow some other thread to proceed */

pthread_cond_signal (cv);   <---- Statement 1
pthread_mutex_unlock (m);
```

# Chapter 16. Using the DCE Threads Exception-Returning Interface

DCE Threads provides the following two ways to obtain information about the status of a threads routine:

- The routine returns a status value to the thread.
- The routine raises an exception.

When you write a multithreaded program, choose only one of the above methods of receiving status. These two methods cannot be used together in the same code module.

The POSIX P1003.4a (pthreads) Draft standard specifies that errors be reported to the thread by setting the external variable *errno* to an error code and returning a function value of -1. This status-value-returning interface is documented in the *z/OS DCE Application Development Reference*. However, an alternative to status values is provided by DCE Threads in the exception-returning interface.

This chapter introduces and provides conventions for the modular use of the exception-returning interface to DCE Threads.

## Syntax for C

Access to exceptions from the C language is defined by the macros in the <**dce/exc_handling.h**> header file. The <**dce/exc_handling.h**> header file is included automatically when you include <**dce/pthread_exc.h**>. See "Using the Exception-Returning Interface" on page 339.

The example in Figure 61 shows the syntax for handling exceptions:

```
TRY
    try_block
[CATCH (exception_name)
    handler_block]...
[CATCH_ALL
    handler_block]
ENDTRY
```

*Figure 61. Syntax for Handling Exceptions*

A **try_block** or a **handler_block** is a sequence of statements, the first of which may be declarations, as in a normal block. If an exception is raised in the **try_block**, the catch clauses are evaluated to see if any one matches the current exception.

The **CATCH** or **CATCH_ALL** clauses absorb an exception. That is, they catch an exception passing out of the **try_block**, and direct it into the associated **handler_block**. Propagation of the exception, by default, then ends. Within the lexical scope of a handler, you can explicitly cause propagation of the same exception to resume (this is called reraising the exception), or you can raise some new exception.

The **RERAISE** statement is allowed in any handler statements and causes the current exception to be reraised. Propagation of the caught exception resumes.

The **RAISE** (**exception_name**) statement is allowed anywhere, and passes a particular exception to a higher level.

```
TRY
    sort(); /* Call a function that may raise an exception.
             * An exception is accomplished by long jumping
             * out of some nested routine back to the TRY
             * clause.  Any output parameters or return values
             * of the called routine are therefore indeterminate.
             */

CATCH (pthread_cancel_e)
    printf("Canceled while sorting\n"); RERAISE;

CATCH_ALL
    printf("Some other exception while sorting\n"); RERAISE;

ENDTRY
```

*Figure 62. Using the RERAISE Statement*

In the preceding example, if the **pthread_cancel_e** exception is raised within the function call, the first **printf** runs. If any other exception is raised within **sort()**, the second **printf** runs. In either situation, passing of the exception to the next level resumes because of the **RERAISE** statement. (If the code is unable to fully recover from the error, or does not understand the error, it needs to do what it did in the previous example and further pass on the error to its callers.)

The following shows the syntax for an epilog:

```
TRY       try_block
[FINALLY  final_block]
ENDTRY
```

The **final_block** runs whether the **try_block** runs to completion without raising an exception, or an exception is raised in the **try_block**. If an exception is raised in the **try_block**, propagation of the exception is resumed after running the **final_block**. if an exception is raised in the **try_block** but is caught either via **CATCH** or **CATCHALL**, the **final_block** is run but the exception is not resumed.

Note that a **CATCH_ALL** handler and **RERAISE** could be used to do this, but the epilog code would then have to be duplicated in two places, as follows:

```
TRY
     try_block
CATCH_ALL
     final_block
     RERAISE;
ENDTRY
{ final_block }
```

A **FINALLY** statement has exactly this meaning, but avoids code duplication.

Another example of the **FINALLY** statement is shown in Figure 63 on page 339:

```
pthread__mutex_lock (some_object.mutex);
some_object.num_waiters = some_object.num_waiters + 1;
TRY
    while (! some_object.data_available)
        pthread_cond_wait (some_object.condition);
    /* The code to act on the data_available goes here */
FINALLY
    some_object.num_waiters = some_object.num_waiters - 1;
    pthread_mutex_unlock (some_object.mutex);
ENDTRY
```

*Figure 63. Example of a FINALLY statement*

In the preceding example, the call to **pthread_cond_wait()** could raise the **pthread_cancel_e** exception. The **final_block** ensures that the shared data associated with the lock is correct for the next thread that acquires the mutex.

## Using the Exception-Returning Interface

To use the exception-returning interface, replace

```
#include <pthread.h>
```

with the following include statement:

```
#include <dce/pthread_exc.h>
```

## Operations on Exceptions

An exception is an object that describes an error condition. Operations on exception objects allow errors to be reported and handled. If an exception is handled correctly, the program can recover from errors. For example, if an exception is raised from a parity error while reading a tape, the recovery action may be to retry 100 times before giving up.

With the DCE Threads Exception-Returning Interface, you can perform the following operations on exceptions:

- Declare and initialize an exception object
- Raise an exception
- Define a region of code over which exceptions are caught
- Catch a particular exception or all exceptions
- Define epilog actions for a block
- Import a system-defined error status into the program as an exception.

These operations are discussed in the following subsections.

## Declaring and Initializing an Exception Object

Declaring and initializing an exception object documents that a program reports or handles a particular error. Having the error expressed as an exception object provides future extensibility as well as portability. Following is an example of declaring and initializing an exception object:

```
EXCEPTION parity_error;         /* Declare it */
EXCEPTION_INIT (parity_error);  /* Initialize it */
```

## Raising an Exception

Raising an exception means reporting an error by passing an exception to a higher level, instead of by returning a value. Passing an exception up to a higher level means searching all the active scopes for code that handles error situations, and running that code. The active scopes are those that are visible to the program. If a scope does not define a handler or epilog block, the scope is simply torn down as the exception is passed on through the stack. This is sometimes called *unwinding the stack*.

A DCE Threads exception cannot resume at the location at which it occurs. It must instead resume at the point where it is caught.

If an exception is not handled, the entire application process is ended. Ending the entire process rather than ending the faulting thread only, cleanly ends the application at the point of error. This prevents the disappearance of the faulting thread from causing problems at some later point.

An example of raising an exception is:

```
RAISE (parity_error);
```

## Defining a Region of Code over Which Exceptions Are Caught

Defining a region of code over which exceptions are caught allows you to call functions that can raise an exception and specify the recovery action.

Following is an example of defining an exception-handling region (without indicating any recovery actions):

```
TRY {
   read_tape ();
    }
ENDTRY;
```

## Catching a Particular Exception or All Exceptions

It is possible to discriminate among errors and perform different actions for each error.

Following is an example of catching a particular exception and specifying the recovery action (in this case, a message). The exception is reraised (passed to its callers) after catching the exception and running the recovery action:

```
TRY {
   read_tape ();
    }
CATCH (parity_error) {
   printf ("Oops, parity error, program ending\n");
   printf ("Try cleaning the heads!\n");
   RERAISE;
   }
ENDTRY
```

*Figure 64. Catching a Particular Exception*

## Defining Epilog Actions for a Block

With a **FINALLY** mechanism, multithreaded programs can restore invariants as certain scopes are unwound, for example, restoring shared data to a correct state and releasing locks. This is often the ideal way to define, in one place, the cleanup activities for a normal or abnormal exit from a block that has changed an invariant.

Figure 65 shows an example of specifying an invariant action whether or not there is an error:

```
lock_tape_drive (t);
TRY
    TRY
       read_tape ();
    CATCH (parity_error)
       printf ("Oops, parity error, program ending\n");
       printf ("Try cleaning the heads!\n");
       RERAISE;
    ENDTRY
    /* Control gets here only if no exception is raised */
    /* Now we can use the data off the tape */
FINALLY
    /* Control gets here normally, or if any exception is raised */
    unlock_tape_drive (t);
ENDTRY
```

*Figure 65. An Example of an Invariant Action*

## Importing a System-Defined Error Status into the Program as an Exception

Most systems define error messages by integer-sized status values. Each status value corresponds to some error message text that should be expressed in the user's own language. The capability to import a status value as an exception permits the DCE Threads Exception-Returning Interface to raise or handle system-defined errors as well as programmer-defined exceptions.

An example of importing an error status into an exception is as follows:

```
exc_set_status (&parity_error, EPARITY);
```

The *parity_error* exception can then be raised and handled like any other exception. If an exception is initialized as an address exception, that is, the only way to reference it is by its address, use **exc_set_status()** to change the exception to *status* type. This renders it representable as a status value as well.

The **exc_raise_status()** call raises exceptions by status value instead of address. In the above example, if a system call fails with errno EPARITY, it is possible to raise an exception even if a specific parity address exception does not exist by calling **exc_raise_status(EPARITY)**.

With the **exc_set_status()** call a CATCH clause can intercept such an exception. The **exc_get_status()** allows a CATCH_ALL to determine the status value representing the problem. The latter is valuable when a specific exception does not exist.

See *z/OS DCE Application Development Reference* for more information on the **exc_set_status()**, **exc_raise_status()**, and **exc_get_status()** calls.

## Rules and Conventions for Modular Use of Exceptions

Adhere to the following rules and conventions to ensure that exceptions are used in a modular way. This enables independent software components to be written without requiring knowledge of each other:

- Use unique names for exceptions.

  A naming convention makes sure that the names for exceptions that are declared **EXTERN** from different modules do not clash. The following convention is recommended:

  ```
  <facility-prefix>_<error_name>_e
  ```

  For example, **pthread_cancel_e**.

- Avoid putting code in a **TRY** routine that belongs before it.

  The **TRY** only guards statements for which the statements in the **FINALLY**, **CATCH**, or **CATCH_ALL** clauses are always valid.

  A common misuse of **TRY** is to put code in the **try_block** that needs to be placed before **TRY**. An example of this misuse is as follows:

  ```
  TRY
      handle = open_file (file_name);
      /* Statements that may raise an exception here */
  FINALLY
      close (handle);
  ENDTRY
  ```

  The preceding **FINALLY** code assumes that no exception is raised by **open_file**. The code accesses an identifier that is not valid in the **FINALLY** part if **open_file** is changed to raise an exception. The preceding example needs to be rewritten as follows:

  ```
  handle = open_file (file_name);
  TRY
      {
      /* Statements that may raise an exception here */
      }
  FINALLY
      close (handle);
  ENDTRY
  ```

  The code that opens the file belongs prior to **TRY**, and the code that closes the file belongs in the **FINALLY** statement. (If **open_file** raises exceptions, it may need a separate **try_block**.)

- Raise exceptions to their proper scope.

  Write functions that pass exceptions to their callers so that the function does not change any persistent process state before raising the exception. A call to the matching **close** call is required only if the **open_file** operation is successful in the current scope.

When **open_file** fails and raises an exception, **close** must not be called because you cannot close a file that is not open. So **open_file** should not be part of the **TRY** clause.

- Do not place a **RETURN** or nonlocal **GOTO** between **TRY** and **ENDTRY** as this destroys the integrity of the exception handling facility and may cause indeterminate problems.

**Important Note to Users**

It is not valid to use **RETURN** or **GOTO**, or to leave by any other means, a **TRY**, **CATCH**, **CATCH_ALL**, or **FINALLY** block. Special code is generated by the **ENDTRY** macro, and it must be run.

- Use the ANSI C volatile attribute.

  Variables that are read or written by exception-handling code must be declared with the ANSI C volatile attribute. Run your tests with the optimize compiler option to ensure that the compiler thoroughly tests your exception-handling code.

- Reraise exceptions that are not fully handled.

  You need to reraise any exception that you catch, unless your handler performs the complete recovery action for the error. This rule permits an unhandled exception to pass on to some final default handler that prints an error message to end the offending thread. (An unhandled exception is an exception for which recovery is incomplete.)

  A corollary of this rule is that **CATCH_ALL** handlers must reraise the exception because they may catch any exception, and usually cannot do recovery actions that are correct for every exception.

  Following this convention is important so that you also do not absorb a cancel or thread-exit request. These are mapped into exceptions so that exception handling has the full power to handle all exceptional conditions from access violations to thread exit. (In some applications, it is important to be able to catch these to work around an erroneously written library package, for example, or to provide a fully fault-tolerant thread.)

## DCE Threads Exceptions and Definitions

Table 16 lists the DCE Threads exceptions and briefly explains the meaning of each exception. Exception names beginning with **exc_** are generic and belong to the exception facility, the underlying system, or both. Exception names beginning with **pthread_** are raised as the result of internal activity in the DCE Threads facility and are not meant to be raised by your code.

*Table 16 (Page 1 of 2). DCE Threads Exceptions*

| Exception | Definition |
|---|---|
| **exc_decovf_e** | An unhandled decimal overflow trap exception occurred. |
| **exc_exquota_e** | The operation failed because of an insufficient quota. |
| **exc_fltdiv_e** | An unhandled floating-point division by zero trap exception occurred. |
| **exc_fltovf_e** | An unhandled floating-point overflow trap exception occurred. |
| **exc_fltund_e** | An unhandled floating-point underflow trap exception occurred. |
| **exc_illaddr_e** | The data or object could not be referred to. |
| **exc_insfmem_e** | There is insufficient virtual storage for the requested operation. |
| **exc_intdiv_e** | An unhandled integer divide by zero trap exception occurred. |
| **exc_intovf_e** | An unhandled integer overflow trap exception occurred.. |
| **exc_nopriv_e** | There is insufficient privilege for the requested operation. |

*Table 16 (Page 2 of 2). DCE Threads Exceptions*

| Exception | Definition |
|---|---|
| **exc_privinst_e** | An unhandled privileged instruction fault exception occurred. |
| **exc_resaddr_e** | An unhandled reserved addressing fault exception occurred. |
| **exc_resoper_e** | An unhandled reserved operand fault exception occurred. |
| **exc_SIGBUS_e** | An unhandled bus error signal occurred. |
| **exc_SIGEMT_e** | An unhandled EMT trap signal occurred. |
| **exc_SIGFPE_e** | An unhandled floating-point exception signal occurred. |
| **exc_SIGILL_e** | An unhandled improper instruction signal occurred. |
| **exc_SIGIOT_e** | An unhandled IOT trap signal occurred. |
| **exc_SIGPIPE_e** | An unhandled broken pipe signal occurred. |
| **exc_SIGSEGV_e** | An unhandled segmentation violation signal occurred. |
| **exc_SIGSYS_e** | An unhandled bad system call signal occurred. |
| **exc_SIGTRAP_e** | An unhandled trace or breakpoint trap signal occurred. |
| **exc_SIGXCPU_e** | An unhandled CPU-time limit exceeded signal occurred. |
| **exc_SIGXFSZ_e** | An unhandled file-size limit exceeded signal occurred. |
| **exc_subrng_e** | An unhandled subscript out-of-range trap exception occurred. |
| **exc_uninitexc_e** | An uninitialized exception was raised. |
| **pthread_badparam_e** | An improper parameter was used |
| **pthread_cancel_e** | A thread cancelation is in progress |
| **pthread_defer_q_full_e** | No space is currently available to process an interrupt request. |
| **pthread_existence_e** | The object referred to does not exist. |
| **pthread_in_use_e** | The object referred to is already in use. |
| **pthread_nostackmem_e** | No space is currently available to create a new stack. |
| **pthread_nostack_e** | The current stack was not created by DCE Threads. |
| **pthread_signal_q_full_e** | Unable to process condition signal from interrupt level. |
| **pthread_stackovf_e** | An attempted stack overflow was detected. |
| **pthread_unimp_e** | This is an unimplemented feature. |
| **pthread_use_error_e** | The requested operation is improperly run. |

To see how to catch z/OS abends as exceptions, see below.

## z/OS ABENDs Caught as Exceptions

The z/OS DCE exception package can catch z/OS system ABENDs. You can catch any abend, or just those that can be mapped to an exception type by using the CATCH_ALL clause. The z/OS DCE exception package is implemented so that all abends that can be caught by the Language Environment Condition Manager are caught by a CATCH_ALL clause. z/OS ABENDs are mapped to a set of exception values. Table 17 on page 345 shows this mapping.

*Table 17. z/OS ABENDs Mapped as OSF Portable Exceptions*

| ABEND | Exception |
|---|---|
| SX'047' | exc_privinst_e |
| SX'0C1' | exc_illinstr_e |
| SX'0C2' | exc_nopriv_e |
| SX'0C3' | exc_resoper_e |
| SX'0C4' | exc_illaddr_e |
| SX'0C5' | exc_resaddr_e |
| SX'0C7' | exc_aritherr_e |
| SX'0C8' | exc_intovf_e |
| SX'0C9' | exc_intdiv_e |
| SX'0CA' | exc_decovf_e |
| SX'0CC' | exc_fltovf_e |
| SX'0CD' | exc_fltund_e |
| SX'0CF' | exc_fltdiv_e |

A coding example that shows how to create and catch z/OS system ABENDs as exceptions is shown in Figure 66.

```
EXCEPTION noload_e;
int       status, reason_code;

EXCEPTION_INIT ( &noload_e );
exc_set_status ( &noload_e, 0X'00806000' );

TRY
   . . .

CATCH (noload_e)         /* handle S806 ABEND */
   . . .

CATCH (exc_illaddr_e)    /* handle S0C4 ABEND */
   . . .

CATCH_ALL                /* handle any other ABEND */

  exc_get_status      ( THIS_CATCH, &status );
  exc_get_reason_code ( THIS_CATCH, &reason_code );

  printf ("Error: ABEND S%03d-%02d occurred!\n", status, reason_code );

ENDTRY
```

*Figure 66. Coding Example: Setting z/OS S806 ABEND to be an Exception*

Exceptions use POSIX signal handlers to process synchronous exceptions and pass control to CATCH processes. Consequently, you can use Language Environment user handlers to get control prior to any

CATCH processing. However, your Language Environment condition handler should always return and propagate the condition to other Language Environment condition handling services. Refer to the Language Environment documentation for more information on Language Environment condition handlers.

## Catching Specific System or User ABENDs

To catch specific z/OS system ABENDs that are not defined in Table 17 on page 345, or to catch user ABENDs (that is, application generated ABENDs), use the following procedure:

1. Dynamically initialize an exception object that has been previously declared using the **EXCEPTION_INIT()** function call. This step defines the exception.
2. Change the exception to a status type using the **exc_set_status()** function call. The object type and status value of the exception are changed. The exception object parameter required for this call is set in the above step.
3. Change the status field to be a user defined type for user or z/OS system abends. These user defined types can be formed by merging the abend code with the reason code as follows.

   The format for the status value is `0X'00sssuuu'`, where

   - sss represents the system abend code
   - uuu represents the user abend code.

   For example, z/OS system abend `SX'047'` can be mapped to a status value of `0X'00470000'`. User abend code `UX'888'` can be mapped to status value `0X'00000888'`. The reason code of an abend is not part of the exception value, thus you must query it separately when you catch an exception using **exc_get_reason_code()**.

**Note:** The status code values in the range of `0X'00FFFFFF'` are reserved for exception abend codes. If you create a status code that is in this range, there is no way for the exception package to distinguish this user code from an abend code.

## Detecting the First Catch of an Exception

To detect the first catch of a specific exception to perform cleanup or a set of specialized function but *only once* during the exception, use the **exc_first_catch()** macro. This method is useful in nested TRY/CATCH clauses. An example of its use is provided in Figure 67 on page 347.

```
TRY  {
     .
     .
     .
}

CATCH ()  {
         exc_first_catch()
         .
         .
         TRY {
             .
             .
             .
         }

         CATCH () {
                 exc_first_catch()
                 .
                 .
                 .
         }

}
```

Figure 67. Detecting the First Catch of an Exception

## Handling Unexpected Exceptions

If an unexpected exception occurs, a dump of diagnostic information is generated by the Language Environment recovery facility under certain conditions. You will receive a message indicating this. Diagnostic information will be logged in the file specified by DD:CEEDUMP. The format of this file is specified by the Language Environment CEEDUMP service. Refer to *z/OS Language Environment Programming Guide*, SA22-7561, for information on the diagnostic information, which includes:

- Program status word (PSW)
- Instruction-length code (ILC) and data in PSW
- z/OS abend code and reason code
- Values in all general registers at time of abend
- Load module name and address of module
- Offset into module
- Stack frame traceback of all called functions and all threads as supplied by Language Environment.
- Values in local variables that are formatted if the application is compiled with proper *test* options.
- Any other Language Environment supplied dump option.

With Language Environment dump services, multiple dumps can be written to the CEEDUMP file as exceptions are raised, or user invoked dumps take place. The dump file is managed by Language Environment and shows the latest dump at the top of the file. Each dump is identified with a title, and the exceptions package provides a title you can use to find out what caused the dump. Following is the dump title format for exceptions:

```
<process>(<correlation_id>)-<exception_keyword>: <exception_value>
```

An example of a dump title for a system abend caught in a CATCH_ALL clause is:

```
EUVSCD(0)-CATCH_ALL: S0C4-04
```

An example of a user abend is:

```
EUVTDTSD(1)-NO TRY:  U4097-08
```

An example of a dump title for a system abend caught in a FINALLY clause is:

```
EUVPCT(0)-FINALLY:   S0D6-04
```

An example of a dump title for not having a specific catch clause for a software exception is:

```
EUVRRPCD(2)-NO CATCH: 0x12345678
```

For exceptions occurring within user processes or applications, the *process* is EUVZUSER.

Determine the cause of the exception by analyzing the dump and then take appropriate action. See *z/OS DCE Messages and Codes* for information on taking appropriate action. If you require information on reading CEEDUMPs, consult *z/OS Language Environment Debugging Guide*, GA22-7560.

Table 18 shows scenarios that lead to a CEEDUMP being taken:

| **Source of Exception** | **No TRY/CATCH Active** | **TRY/CATCH Active** **Exception Uncaught** | **TRY/CATCH Active** **Exception Caught by CATCH** | **TRY/CATCH Active** **Exception caught by CATCH_ALL or FINALLY** |
|---|---|---|---|---|
| Software Exception Raised (RAISE) | NO DUMP | No DUMP | NO DUMP | NO DUMP |
| Software SIGNAL Raised (raise) | NO DUMP | NO DUMP | NO DUMP | NO DUMP |
| System Abend | Exception DUMP in handler | Exception DUMP in handler | NO DUMP | Exception DUMP in handler |
| User Abend | Exception DUMP in handler | Exception DUMP in handler | NO DUMP | Exception DUMP in handler |
| RERAISED Exception | NO DUMP | NO DUMP | NO DUMP | NO DUMP |

*Table 18. Exception Dump Scenarios*

When a dump occurs, the Exceptions component queries the Reliability, Availability, and Serviceability (RAS) services component to determine if a symptom string is logged to identify the dump. If required, the RAS services are notified so that a symptom string defined in the catch area is attached to the dump. If no catch is made for the exception, a generic symptom string is attached by the Exceptions component to the dump.

# Chapter 17. DCE Threads Example

The example in this chapter shows the use of DCE Threads in a C program that performs a prime number search. The program finds a specified number of prime numbers, and then sorts and displays these numbers. Several threads participate in the search. Each thread:

1. Takes a number (the next one to be checked)
2. Checks if the number is a prime number
3. Records the number if it is prime
4. Takes another number, and so on.

This program shows the work crew model of programming (described in “Work Crew Model” on page 316). The workers (threads) increment a number (**current_num**) to get their next work assignment, which, in this case, is the same task as before but with a different number to check for a prime. As a whole, the worker threads are responsible for finding a specified number of prime numbers, at which point their work is completed.

## Details of Program Logic and Implementation

The number of workers to be used and the requested number of prime numbers to be found are defined constants. A macro is used to check for bad status (bad status returns a value of -1), and to print a given string and the associated error value upon bad status. Data to be accessed by all threads (mutexes, condition variables, and so forth) is declared as global items.

Worker threads execute the prime search routine, which begins by synchronizing with the boss (or parent) thread using a predicate and a condition variable. Always enclose a condition wait in a predicate loop to prevent a thread from continuing if it receives a spurious wake up. The lock associated with the condition variable must be held by the thread when the condition wait call is made. The lock is implicitly released within the condition wait call and acquired again when the thread resumes. The same mutex must be used for all operations performed on a specific condition variable.

After the parent sets the predicate and broadcasts, the workers begin finding prime numbers until canceled by a fellow worker who has found the last requested prime number. Upon each iteration, the workers increment the current number and take the new value as their work item. A mutex is locked and unlocked around the operation to get the next work item. The purpose of the mutex is to ensure the completeness and integrity of this operation, and the visibility of the new value across all threads. This type of locking protocol needs to be performed on all global data to ensure its visibility and protect its integrity.

Each worker thread then determines if its current work item (a number) is prime by trying to divide numbers into it. If the number proves to be indivisible, it is put on the list of primes. Cancel operations are explicitly turned off while working with the list of primes to better control any cancelations that do occur. The list and its current count are protected by locks, which also protect the cancelation process of all other worker threads upon finding the last requested prime. While still under the prime list lock, the current worker checks to see if it has found the last requested prime, and if so, unsets a predicate and cancels all other worker threads. Cancels are then reenabled. The canceling thread falls out of the work loop as a result of the predicate that it unsets.

The parent thread’s flow of execution is as follows: set up the environment, create worker threads, broadcast to them that they can start, join each thread as it finishes, and sort and print the list of primes.

- Setting up of the environment requires initializing mutexes and the one condition variable used in the example.
- Creation of worker threads is straightforward and utilizes the default attributes (**pthread_attr_default()**). Notice again that locking is performed around the predicate on which the condition variable wait loops. In this case, the locking is simply done for visibility and is not related to the broadcast function.
- As the parent joins each of the returning worker threads, it receives an exit value from them that indicates whether a thread exited normally or not. In this case, the exit values on all but one of the worker threads are -1, indicating that they were canceled.
- The list is then sorted to ensure that the prime numbers are in order from lowest to highest.

The following pthread routines are used in this example:

- **pthread_cancel()**
- **pthread_cond_broadcast()**
- **pthread_cond_init()**
- **pthread_cond_wait()**
- **pthread_create()**
- **pthread_detach()**
- **pthread_exit()**
- **pthread_join()**
- **pthread_mutex_init()**
- **pthread_mutex_lock()**
- **pthread_mutex_unlock()**
- **pthread_setcancel()**
- **pthread_testcancel()**

## Threads Example — Searching for Prime Numbers

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
/*
 * Constants used by the example.
 */
#define     workers     5          /* Threads to perform prime check  */
#define     request     110        /* Number of primes to find        */
/*
 * Macros
 */
#define check(status,string)     if (status == -1) perror (string)
/*
 * Global data
 */
pthread_mutex_t prime_list;     /* Mutex for use in accessing the prime     */
pthread_mutex_t current_mutex;  /* Mutex associated with current number     */
pthread_mutex_t cond_mutex;     /* Mutex used for ensuring CV integrity     */
pthread_cond_t  cond_var;       /* Condition variable for thread start      */
int             current_num= -1;/* Next number to be checked, start odd     */
int             thread_hold= 1; /* Number associated with condition state   */
int             count=0;        /* Count of prime numbers - index to primes */
int             primes[request];/* Store prime numbers - synchronize access */
pthread_t       threads[workers];/* Array of worker threads                 */
/*
 * Worker thread routine.
 *
 * Worker threads start with this routine, which begins with a condition
 * wait designed to synchronize the workers and the parent.  Each worker
 * thread then takes a turn taking a number for which it will determine
 * whether or not it is prime.
 *
 */
void
prime_search (pthread_addr_t arg)
    {
    div_t   div_results;           /* DIV results: quot and rem       */
    int     numerator;             /* Used for determining primeness   */
    int     denominator;           /* Used for determining primeness   */
    int     cut_off;               /* Number being checked div 2       */
    int     notifiee;              /* Used during a cancelation        */
    int     prime;                 /* Flag used to indicate primeness  */
    int     my_number;             /* Worker thread identifier         */
    int     status;                /* Hold status from pthread calls   */
    int     not_done=1;            /* Work loop predicate              */
    my_number = (int)arg;
```

*Figure 68 (Part 1 of 5). Threads Example Searching*

```
    /*
     * Synchronize threads and the parent using a condition variable, for
     * which the predicate (thread_hold) will be set by the parent.
     */
    status = pthread_mutex_lock (&cond_mutex);
    check(status,"1:Mutex_lock bad status\n");

    while (thread_hold) {
        status = pthread_cond_wait (&cond_var, &cond_mutex);
        check(status,"3:Cond_wait bad status\n");
        }

    status = pthread_mutex_unlock (&cond_mutex);
    check(status,"4:Mutex_unlock bad status\n");
    /*
     * Perform checks on ever larger integers until the requested
     * number of primes is found.
     */
    while (not_done) {

        /* cancelation point */
        pthread_testcancel ();

        /* Get next integer to be checked */
        status = pthread_mutex_lock (&current_mutex);
        check(status,"6:Mutex_lock bad status\n");

        current_num = current_num + 2;              /* Skip even numbers */
        numerator = current_num;

        status = pthread_mutex_unlock (&current_mutex);
        check(status,"9:Mutex_unlock bad status\n");

        /* Only need to divide in half of number to verify not prime */
        cut_off = numerator/2 + 1;
        prime = 1;

        /* Check for prime; exit if something evenly divides */
        for (denominator = 2; ((denominator < cut_off) && (prime));
                                                    denominator++) {
            prime = numerator % denominator;
            }
        if (prime != 0) {

            /* Explicitly turn off all cancels */
            pthread_setcancel(CANCEL_OFF);
```

*Figure 68 (Part 2 of 5). Threads Example Searching*

```
            /*
             * Lock a mutex and add this prime number to the list. Also,
             * if this fulfills the request, cancel all other threads.
             */
            status = pthread_mutex_lock (&prime_list);
            check(status,"10:Mutex_lock bad status\n");

            if (count < request)  {
                primes[count] = numerator;
                count++;
                }
            else if (count == request) {
                not_done = 0;
                count++;
                for (notifiee = 0; notifiee < workers; notifiee++) {
                    if (notifiee != my_number) {
                        status = pthread_cancel ( threads[notifiee] );
                        check(status,"12:Cancel bad status\n");
                        }
                    }
                }

            status = pthread_mutex_unlock (&prime_list);
            check(status,"13:Mutex_unlock bad status\n");

            /* Reenable cancels */
            pthread_setcancel(CANCEL_ON);
            }
        pthread_testcancel ();
        }
    pthread_exit ((pthread_addr_t) my_number);
    }
int main()
    {
    int     worker_num;     /* Counter used when indexing workers   */
    int     exit_value;     /* Individual worker's return status    */
    int     list;           /* Used to print list of found primes   */
    int     status;         /* Hold status from pthread calls       */
    int     index1;         /* Used in sorting prime numbers        */
    int     index2;         /* Used in sorting prime numbers        */
    int     temp;           /* Used in a swap; part of sort         */
    int     not_done;       /* Indicates swap made in sort          */

    setlocale(LC_ALL, "");
```

*Figure 68 (Part 3 of 5). Threads Example Searching*

```
     /*
      * Create mutexes
      */
    status = pthread_mutex_init (&prime_list, pthread_mutexattr_default);
    check(status,"15:Mutex_init bad status\n");
    status = pthread_mutex_init (&cond_mutex, pthread_mutexattr_default);
    check(status,"16:Mutex_init bad status\n");
    status = pthread_mutex_init (&current_mutex, pthread_mutexattr_default);
    check(status,"17:Mutex_init bad status\n");

    /*
     * Create conditon variable
     */
    status = pthread_cond_init (&cond_var, pthread_condattr_default);
    check(status,"45:Cond_init bad status\n");

    /*
     * Create the worker threads.
     */
    for (worker_num = 0; worker_num < workers; worker_num++) {
        status = pthread_create (
            &threads[worker_num],
            pthread_attr_default,
            (pthread_startroutine_t)prime_search,
            (pthread_addr_t)worker_num);
        check(status,"19:Pthread_create bad status\n");
    }

    /*
     * Set the predicate thread_hold to zero, and broadcast on the
     * condition variable that the worker threads may proceed.
     */
    status = pthread_mutex_lock (&cond_mutex);
    check(status,"20:Mutex_lock bad status\n");

    thread_hold = 0;

    status = pthread_cond_broadcast (&cond_var);
    check(status,"20.5:cond_broadcast bad status\n");

    status = pthread_mutex_unlock (&cond_mutex);
    check(status,"21:Mutex_unlock bad status\n");
```

*Figure 68 (Part 4 of 5). Threads Example Searching*

```
    /*
     * Join each of the worker threads in order to obtain their
     * summation totals, and to ensure each has completed
     * successfully.
     *
     * Mark thread storage free to be reclaimed upon termination by
     * detaching it.
     */
    for (worker_num = 0; worker_num < workers; worker_num++) {

        status = pthread_join (
            threads[worker_num],
            (void**)&exit_value );
        check(status,"23:Pthread_join bad status\n");

        if (exit_value == worker_num) printf("thread terminated normally\n");

        status = pthread_detach ( &threads[worker_num] );
        check(status,"25:Pthread_detach bad status\n");
        }

   /*
    * Take the list of prime numbers found by the worker threads and
    * sort them from lowest value to highest.  The worker threads work
    * concurrently; there is no guarantee that the prime numbers
    * will be found in order. Therefore, a sort is performed.
    */
    not_done = 1;
    for (index1 = 1; ((index1 < request) && (not_done)); index1++) {
        for (index2 = 0; index2 < index1; index2++) {
            if (primes[index1] < primes[index2]) {
                temp = primes[index2];
                primes[index2] = primes[index1];
                primes[index1] = temp;
                not_done = 0;
                }
            }
        }

    /*
     * Print out the list of prime numbers that the worker threads
     * found.
     */
    printf ("The list of %d primes follows:\n", request);
    printf("%d",primes[0]);

    for (list = 1; list < request; list++) {
        printf (",%d", primes[list]);
        }

    printf ("\n");

    return(0);
    }
```

*Figure 68 (Part 5 of 5). Threads Example Searching*

# Chapter 18. Comparing POSIX Multithreading to z/OS Multitasking

Although the z/OS operating system has the ability to dispatch multiple threads of execution, it is different from the POSIX definition. To compare the two, you must first understand the meaning of some key C and POSIX constructs.

**Heap**
The area of memory from which dynamic storage is allocated using the ANSI memory allocation routines like **malloc()**, **calloc()**, **free()**, and so on.

**Static Area**
The area of memory where global variables (variables declared outside the scope of the **main()** program or outside the scope of any subroutines) and string constants (string values that appear within double quotation marks within a program) are stored. Variables that are declared **static** *within* the scope of a subroutine are also stored in this area.

**Stack**
The area of memory used for parameter passing and for storing automatic variables (variables defined within the scope of **main()** program or any subroutine).

**Process**
A single program that consists of one **main()** program and any number of subroutines. Each process has its own heap and static area.

**Thread**
A subroutine dispatched as a separate executable entity from the **main()** program (which itself is also a thread).

**task**
A TASK (in z/OS) is a unit of execution. POSIX threads, as used by DCE, are mapped to z/OS tasks (TCBs). These mappings occur in two flavors:

- heavy weight

  **pthread** creation is the same as task creation, and **pthread** deletion is the same as task deletion (TCB DETACH). z/OS resource managers are run.

- medium weight

  **pthread** creation is an *association* of a thread with an existing z/OS TCB on a one-for-one basis. **pthread** deletion ends this association; the z/OS task is not DETACHed and z/OS resource managers are not run.

DCE uses medium weight threads.

The key distinction between z/OS *tasking* and POSIX *threading* is that each z/OS task is close to being a separate process, whereas each POSIX thread is actually a program subroutine. Refer to Figure 69 on page 358 for a comparison of threads to processes. In the following sections, the term *thread* refers to the POSIX execution entity, while the term *task* refers to the z/OS execution entity.

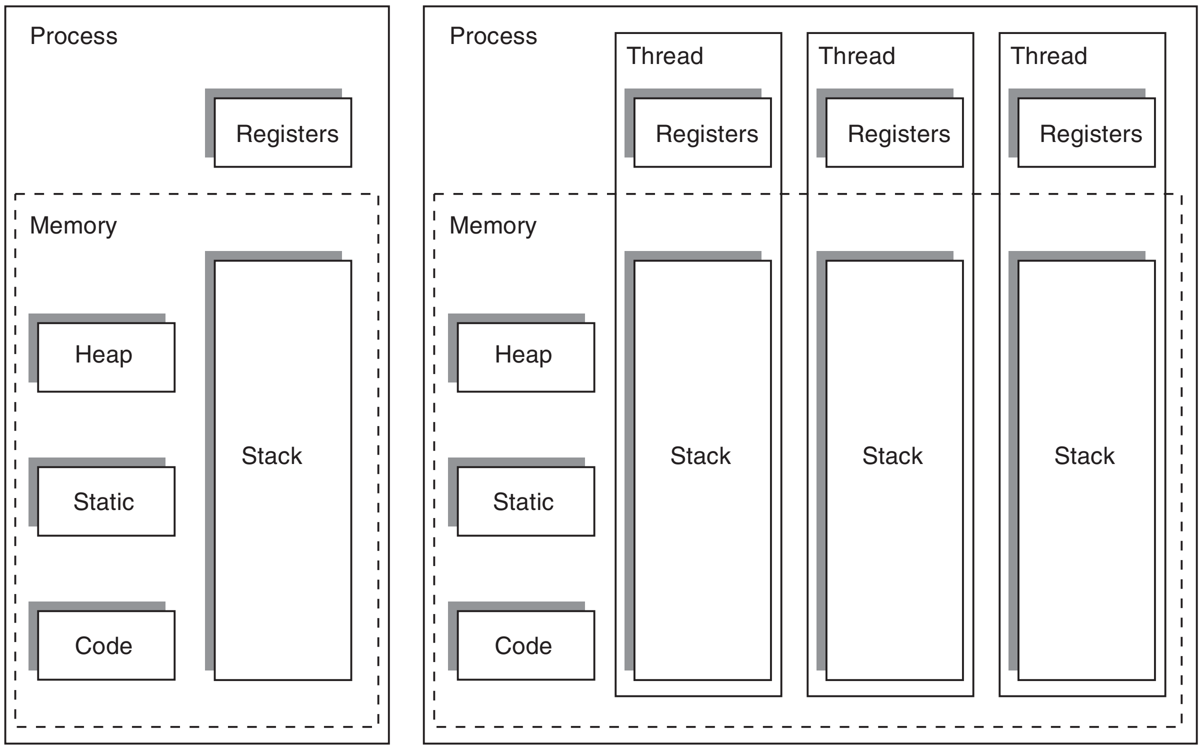


*Figure 69. Threading Model Overview*

Although a z/OS task is more easily mapped to a process, it is still necessary to associate a POSIX thread with a z/OS task. The z/OS DCE threading services map each thread to a separate Task Control Block (TCB), but do not change the semantics of UNIX System Services Threads. A TCB is a data area used to store information regarding a z/OS operating system dispatchable unit called a task.

## Types of Threads

The z/OS DCE threading service defines two types of threads:

**Heavy-weight** This type of thread is given a new TCB each time one is created. Thus, there is an exact mapping between threads and tasks.

**Medium-weight** This type of thread is subdispatched, meaning that dispatching is controlled within the threads implementation itself. (Task dispatching is still handled by z/OS.) A pool of TCBs is used where any TCB may be used as the means to execute the thread. There is a configurable maximum limit of TCBs as well as a configurable ratio of TCBs to threads (because you can have more threads than TCBs). Because threads are subdispatched using a pool of TCBs, multiple threads may share the same TCB.

If for any reason the TCB abnormally terminates (ABENDs) while executing a thread, the recovery exit (for example, ESTAE or ESPIE) does not distinguish between threads. Consequently, the TCB will indiscriminately clear up resources, including resources that are still being used by a thread running under a different TCB.

The type of thread is selectable based on a field within the thread attribute (which is created by the caller). The default for DCE Threads is to use medium-weight threads. For UNIX System Services Threads, which are described in *z/OS C/C++ Run-Time Library Reference*, SA22-7821, the default is heavy-weight threads.

**Note to Readers**

For z/OS DCE, this field is set internally to medium-weight threads. To change the thread weight, see the description of the **pthread_attr_setweight_np()** API in *z/OS C/C++ Run-Time Library Reference*, SA22-7821.

## Choosing the Type of Thread

**Note:** This section does not apply to manager Threads, that is, the threads created by RPC in DCE server applications to handle incoming RPCs. Most thread attributes are set in the attributes object and cannot be changed dynamically by the manager code. This section only applies to threads you create using **pthread_create()**.

You must consider the following when choosing the type of thread:

1. Desired performance characteristics.

   Medium-weight threads minimize both the dispatch time (the time it takes to search the thread queue to find a ready thread, reset the machine registers and initiate its execution) as well as the context switch time (the time taken to put one thread back onto the dispatch queue and dispatch another thread).

2. The z/OS recovery characteristics of a thread.

   Heavy-weight threads should be used when issuing z/OS subsystem calls (for example, calls to Virtual Telecommunications Access Method (VTAM®) or DATABASE 2™ (DB2®)). The reason is that subsystems will often attach cleanup exits that they expect to be driven when a task terminates. Because medium-weight threads are subdispatched using a pool of TCBs, multiple threads may share the same TCB.

# Chapter 19. Migrating between UNIX System Services and DCE Threads

The DCE Threads package is based on the pthreads interface specified in the POSIX 1003.4a, Draft 4 standard. The z/OS UNIX System Services Threads package is based on the POSIX 1003.4a, Draft 6 standard. Throughout this chapter, the z/OS UNIX System Services Threads package is referred to as *UNIX System Services Threads*, whereas the DCE Threads package is referred to as *DCE Threads*.

There are two sets of Threads services to choose from because the POSIX Threads standard, 1003.4a, is an evolving standard. DCE Threads captures the Threads services at the Draft 4 level, and z/OS UNIX System Services captures the Threads services at the Draft 6 level.

The reasons for using the two types of threading packages are summarized in Table 19.

| DCE Threads | UNIX System Services Threads |
|---|---|
| • Portability of DCE applications, written on other DCE platforms, to and from the z/OS platform. DCE implementations are most likely to support the *standard* DCE threads interfaces.<br>• Non-POSIX extensions. DCE Threads offers a number of services that are not in the UNIX System Services Threads suite:<br>– **_np** (non-portable) services<br>– Exception handling (see Chapter 16, “Using the DCE Threads Exception-Returning Interface” on page 337)<br>– Three types of mutex objects. | Greater evolution of the Threads standard |

*Table 19. Reasons for Using DCE or UNIX System Services Threads*

If you are programming with DCE threads, refer to the earlier chapters contained in Part 3, “Using the DCE Threads APIs” on page 313. If you are programming with UNIX System Services threads, refer to the threading information in *z/OS C/C++ Run-Time Library Reference*, SA22-7821. Note that for consistency, you should not mix draft 4 and draft 6 in your source code. You can, however, code a DCE application using draft 6, and link it using the DCE library which is based on draft 4. The following information highlights the differences between the two packages, and assists you if you are migrating your multithreaded application between the them.

## Differences between UNIX System Services and DCE Threads

There are three major differences between the two packages:

- Signals

  DCE threads implements per-thread signal handling for synchronous signals. UNIX System Services threads does not. See “Using Signals” on page 331 for a detailed discussion of DCE-style signal handling.

- Mutexes

  DCE threads offers three types of mutexes (see “Mutexes” on page 323) and a global lock (see “Global Lock” on page 335). UNIX System Services threads offers two types of mutexes and no global lock.

- Programming Interfaces

  There are semantic and syntactic differences of related pthread API calls between the two packages.

## Choosing DCE or UNIX System Services Threads

To use either of the above Threads packages, you must include either the **pthread.h** header file or the **dce/pthread_exc.h** header file. The **dce/pthread_exc.h** header file enables the exception returning interface as well as the **pthread** interface.

The **pthread.h** file is included as follows:

```
#include <pthread.h>
```

The **dce/pthread_exc.h** file is included as follows:

```
#include <dce/pthread_exc.h>
```

To use the DCE Threads package, define the C language macro constant **_DCE_THREADS**. You can define the constant by one of the following methods:

- Specifying a compile-time definition
- Defining the constant in the source file prior to the inclusion of the **pthread.h** header file in your source.

To use the UNIX System Services threads package, do not define the **_DCE_THREADS** macro constant. Note that if you use the UNIX System Services threads package, you cannot use the DCE threads APIs that return exceptions.

## Mutexes

A detailed discussion of the three types of DCE Threads mutexes is found in “Mutexes” on page 323. Briefly, the three types are fast, recursive, and nonrecursive. UNIX System Services Threads supports two types: recursive and nonrecursive.

For z/OS DCE, the DCE Threads *fast* mutex is equivalent to nonrecursive. In other words, the fast mutex does not exist; identifying a fast mutex to **pthread_mutex_setkind_np()** results in a nonrecursive mutex.

## Differences between DCE Threads and UNIX System Services Threads

If your multithreaded programming application uses the DCE threads APIs based on the POSIX 1003.4 Draft 4 standard, and you want to run it using the UNIX System Services Threads package available on z/OS, be aware of the following semantic and syntactic differences between the two threading packages. You must change your application accordingly. Conversely, if your multithreaded application has been written for the Draft 6 standard and you want to run it using the Draft 4 standard, you must reverse those changes.

The following information is a summary of the differences. For a complete understanding of the UNIX System Services Threads package, refer to *z/OS C/C++ Run-Time Library Reference*, SA22-7821, and for DCE threads APIs refer to *z/OS DCE Application Development Reference*.

## Changes to Threads APIs

The following DCE Threads APIs have been changed (in their entirety) to their corresponding UNIX System Services Threads APIs. Thus, if you use one of the following DCE Threads APIs in your application, and you want to migrate to the UNIX System Services Threads, replace the DCE Threads API with the corresponding UNIX System Services API in the following list:

| DCE Threads API | UNIX System Services Threads API |
|---|---|
| **pthread_setcancel()** | **pthread_setintr()** |
| **pthread_setasynccancel()** | **pthread_setintrtype()** |
| **pthread_testcancel()** | **pthread_testintr()** |
| **pthread_attr_create()** | **pthread_attr_init()** |
| **pthread_attr_delete()** | **pthread_attr_destroy()** |
| **pthread_condattr_create()** | **pthread_condattr_init()** |
| **pthread_condattr_delete()** | **pthread_condattr_destroy()** |
| **pthread_mutexattr_create()** | **pthread_mutexattr_init()** |
| **pthread_mutexattr_delete()** | **pthread_mutexattr_destroy()** |
| **pthread_keycreate()** | **pthread_key_create()** |
| **pthread_yield()** | **pthread_yield(NULL)** |

## Specifying Attributes Objects

The attribute parameter on the DCE Threads version of the following calls have changed to a pointer on the UNIX System Services Threads version of the calls:

- **pthread_create()**
- **pthread_cond_init()**
- **pthread_mutex_init()**.

In addition, other DCE Threads attribute APIs such as **pthread_attr_getstacksize()** have been changed to return attribute values in output buffers rather than by function return.

## Call Attributes Not Supported by UNIX System Services Threads

The following DCE Threads call attributes are not supported by UNIX System Services Threads:

- **pthread_attr_default**
- **pthread_mutexattr_default**
- **pthread_condattr_default**.

When using UNIX System Services Threads, specify default attributes with NULL where an attribute object is required.

## Types Not Supported by UNIX System Services Threads

The following types are not available on UNIX System Services Threads:

- **pthread_startroutine_t**
- **pthread_initroutine_t**
- **pthread_cleanup_t**
- **pthread_destructor_t**
- **pthread_addr_t**.

## Mutex Types

The following DCE Threads mutex types are redefined for the UNIX System Services Threads package:

| DCE Threads mutex type | UNIX System Services Threads mutex type |
|---|---|
| **MUTEX_RECURSIVE** | **__MUTEX_RECURSIVE** |
| **MUTEX_RECURSIVE_NP** | **__MUTEX_RECURSIVE** |
| **MUTEX_FRIENDLY** | **__MUTEX_RECURSIVE** |
| **MUTEX_NON_RECURSIVE** | **__MUTEX_NONRECURSIVE** |
| **MUTEX_NON_RECURSIVE_NP** | **__MUTEX_NONRECURSIVE** |
| **MUTEX_NONRECURSIVE_NP** | **__MUTEX_NONRECURSIVE** |
| **MUTEX_FAST_NP** | **__MUTEX_NONRECURSIVE** |

## Cancelability Versus Interruptibility

The DCE Threads notion of cancelability and the constants **CANCEL_ON** and **CANCEL_OFF** is not available in the UNIX System Services Threads. In its place is the notion of interruptability and the following constants:

- **PTHREAD_INTR_ENABLE**
- **PTHREAD_INTR_DISABLE**
- **PTHREAD_INTR_CONTROLLED**
- **PTHREAD_INTR_ASYNCHRONOUS**.

## Semantic Differences

Be aware of the following semantic differences between the two threads packages:

- OSF DCE Threads allows multiple callers of the **pthread_join()** targeting a single thread. z/OS DCE Threads and UNIX System Services Threads allow only one thread to join another thread.
- The UNIX System Services Threads API **pthread_setintr()** is an interruption point. The related DCE Threads API **pthread_setcancel()** is *not* a cancelation point.
- The default thread type for DCE Threads is medium-weight threads while the default thread type for UNIX System Services Threads is heavy-weight threads.

In addition to the above, there is a syntactic difference with **pthread_mutex_trylock()** call in that the UNIX System Services Threads and DCE Threads return codes for this call are different.

## Miscellaneous Differences

In addition to the above, be aware of the following differences between DCE Threads and UNIX System Services Threads:

- The DCE Threads **pthread_once_init** macro constant is known in the UNIX System Services Threads package as the **PTHREAD_ONCE_INIT** macro constant.
- For the **pthread_yield()** API call, the parameter in the DCE Threads package is `void`, while in UNIX System Services Threads it is `void *`, and NULL is expected.

# Part 4. Using the DCE Distributed Time Service APIs

This part of the book shows you how to use the DTS APIs in DCE applications to translate among different time formats and representations to determine event sequencing, duration, and scheduling. In addition, you are shown how to use external time-provider services with the DCE Time-Provider Interface. The final chapter presents a programming example showing you how to use the DTS APIs.

# Chapter 20. Introduction to the Distributed Time Service API

This chapter describes the DCE Distributed Time Service (DTS) programming routines. You can use these routines to obtain timestamps that are based on Coordinated Universal Time (UTC). You can also use the DTS routines to translate among different timestamp formats and perform calculations on timestamps. Applications can use the timestamps that DTS supplies to determine event sequencing, duration, and scheduling. Applications can call the DTS routines from server or clerk systems.

The DCE DTS routines are written in the C programming language. You should be familiar with the basic DTS concepts before you attempt to use the Applications Programming Interface (API). The DTS chapters of *z/OS DCE Administration Guide* provides conceptual information about DTS.

The DTS API routines offer the following basic functions:

- Retrieving timestamp information
- Converting from a binary timestamp and other timestamps that use different time structures
- Converting from a binary timestamp and text string representation
- Converting between UTC time and local time
- Manipulating binary timestamps
- Comparing two binary time values
- Calculating binary time values
- Obtaining time zone information.

**Note:** For information on the relationship between the DCE DTS and synchronization of System/390® and **z**Series 900 hardware clocks, including the 9037 External Time Reference feature used by z/OS systems, refer to *z/OS DCE Administration Guide*.

The following sections describe how DTS represents time, discuss the DTS time structures and the DTS API header files, and briefly describe the DTS API routines.

## DTS Time Representation

Coordinated Universal Time (UTC) is the international time standard that has largely replaced Greenwich Mean Time (GMT). The standard is administered by the International Time Bureau (BIH), and is widely used. DTS uses opaque binary timestamps that represent UTC for all of its internal processes. You cannot read or disassemble a DTS binary timestamp; with the DTS API, applications can convert or manipulate timestamps, but cannot display them directly. However, DTS also translates the binary timestamps into text strings, which can be displayed.

## Absolute Time Representation

An ***absolute time*** is a point on a time scale. For DTS, absolute times refer to the UTC time scale; absolute time measurements are derived from DCE system clocks or external time-providers. When DTS reads a DCE system clock time, it records the time in an opaque binary timestamp that also includes the inaccuracy and other information. When you display an absolute time, DTS converts the time to a text string as shown in the following display:

```
1990-11-21-13:30:25.785-04:00I000.082
```

DTS displays all times in a format that complies with the International Organization for Standardization (ISO) 8601 (1988) standard.

**Note:** The inaccuracy portion of the time is not defined in the ISO standard. (Times that do not include an inaccuracy are accepted.) Figure 70 explains the ISO format that generated the previous display.

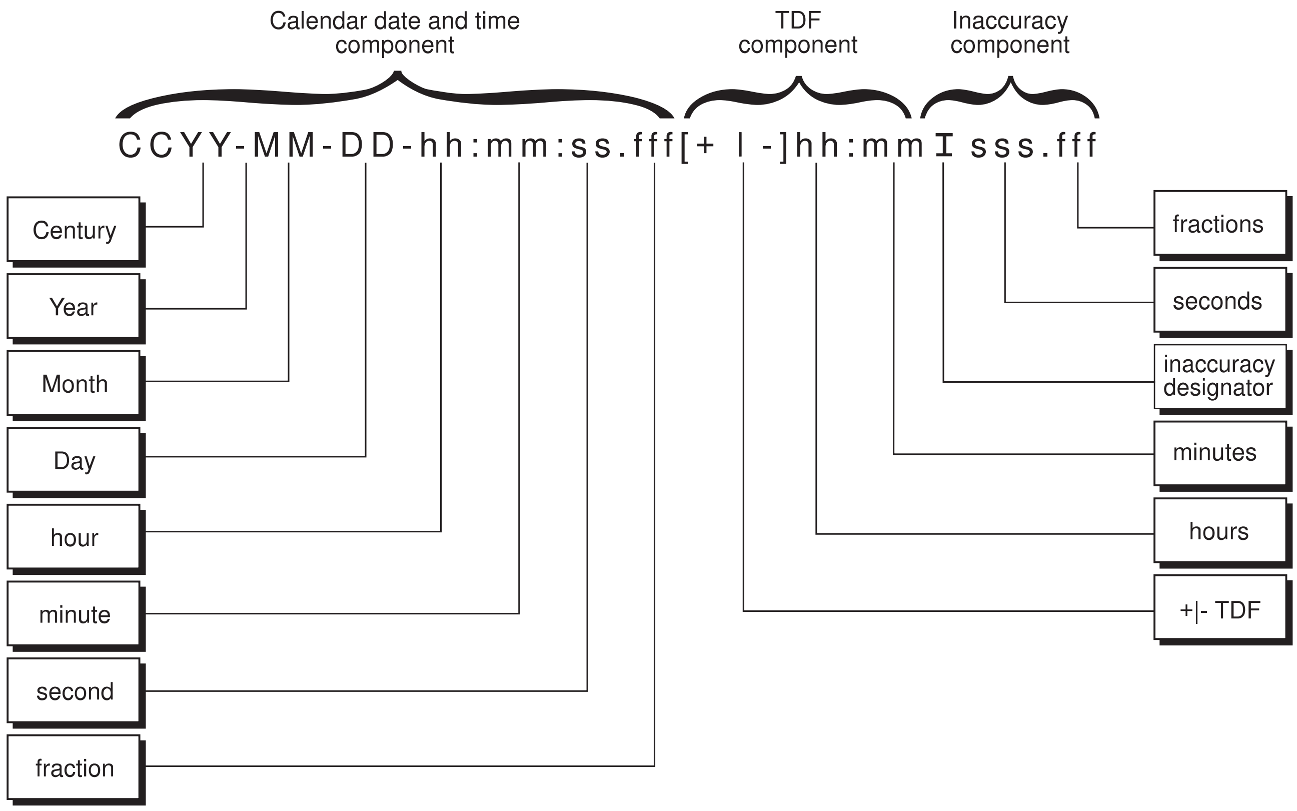


*Figure 70. ISO Format for Time Displays*

The relative time preceded by the plus (+) or minus (-) character indicates the hours and minutes that the calendar date and time are offset from UTC. The presence of this ***Time Differential Factor*** (TDF) in the string also indicates that the calendar date and time are the local time of the system, not UTC. Local time is UTC plus the TDF. The Inaccuracy designator **I** indicates the beginning of the inaccuracy component associated with the time.

Although DTS displays all times in the previous format, variations to the ISO format shown in Figure 71 on page 371 are also accepted as input for the text string conversion routines.

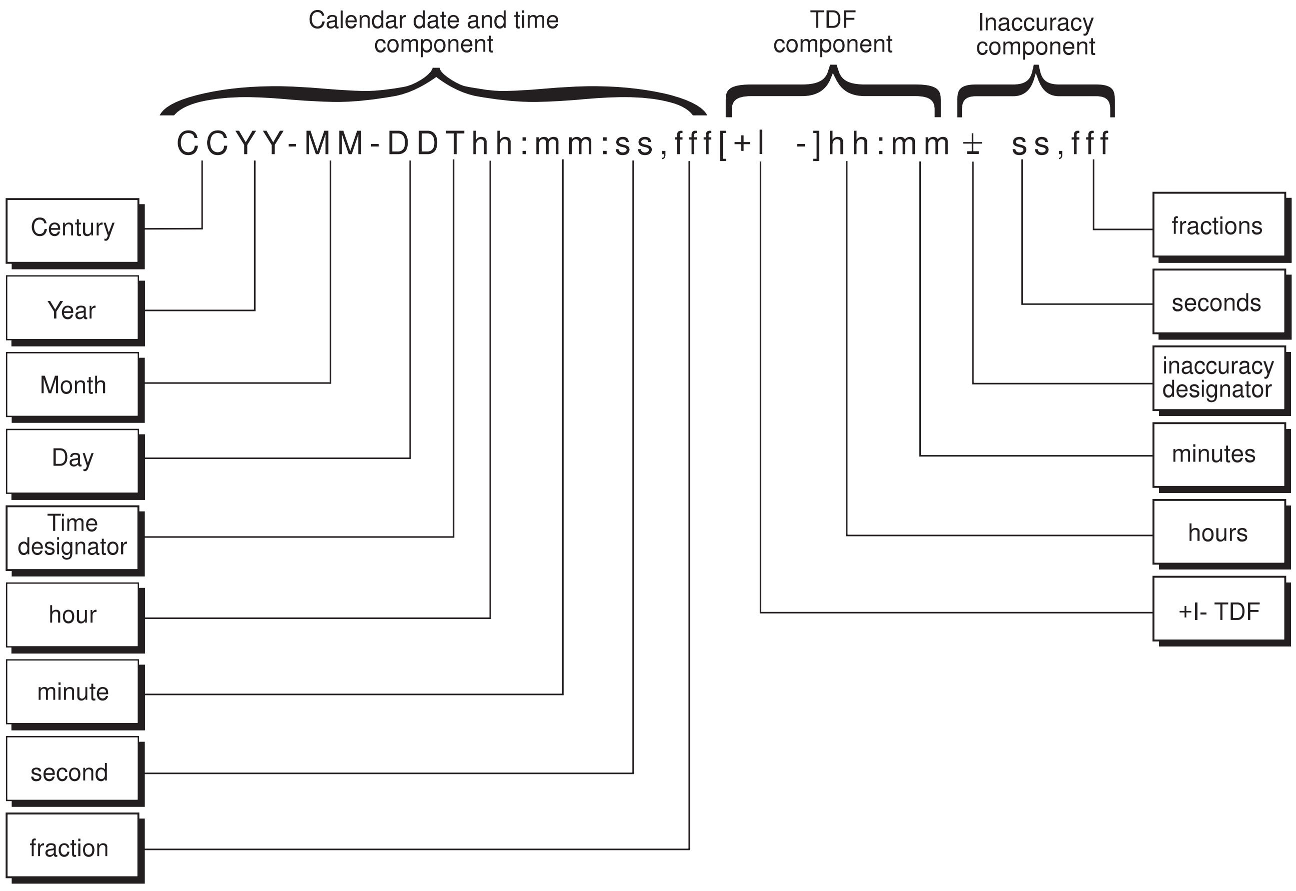


*Figure 71. Changed ISO Format acceptable as Input*

The Time designator, **T**, separates the calendar date from the time, a **,** (comma), separates seconds from fractional seconds, and the **+** (plus) or **-** (minus) character indicates the beginning of the Time Differential Factor (TDF) or the inaccuracy component.

**Examples of Valid Time Formats:** The following represents July 4, 1776 17:01 GMT and an unspecified inaccuracy (default).

```
1776-7-4-17:01:00
```

The following represents a local time of 12:01 (17:01 GMT) on July 4, 1776 with a TDF of -5 hours and an inaccuracy of 100 seconds.

```
1776-7-4-12:01:00-05:00I100
```

Both of the following represent 12:00 GMT in the current day, month, and year with an unspecified inaccuracy.

```
12:00 and T12
```

The following represents July 14, 1792 00:00 GMT with an unspecified inaccuracy.

```
1792-7-14
```

## Relative Time Representation

A ***relative time*** is a discrete time interval that is usually added to, or subtracted from, another time. A TDF associated with an absolute time is one example of a relative time. A relative time is normally used as input for commands or system routines.

Figure 72 shows the full syntax for a relative time.

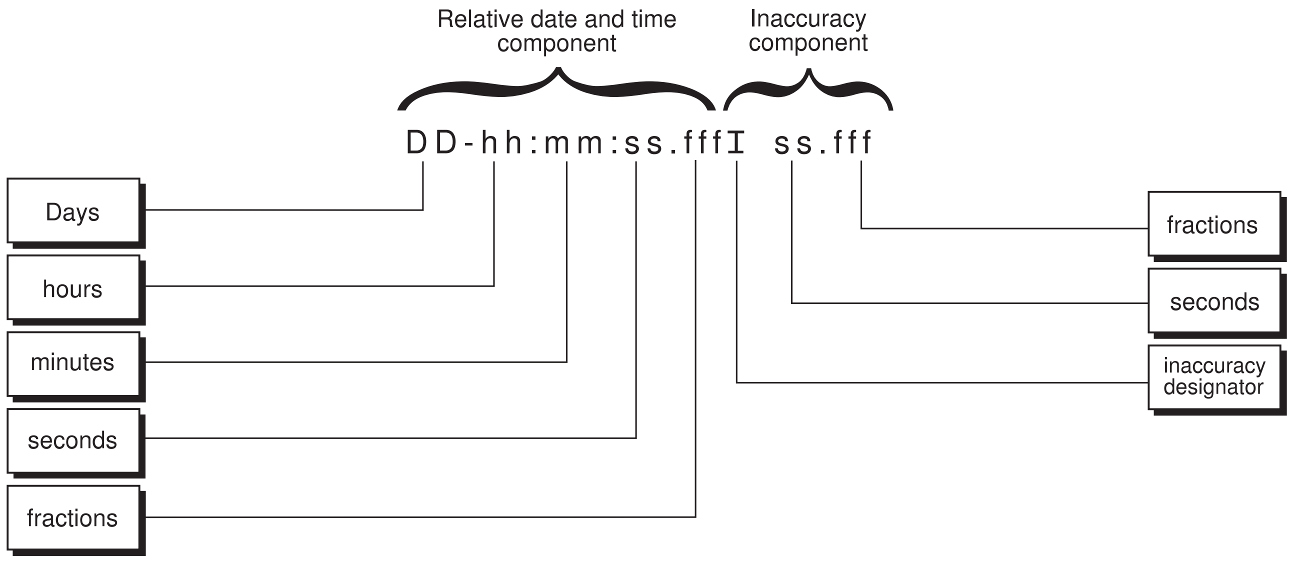


*Figure 72. Full Syntax for a Relative Time*

The following example shows a relative time of 21 days, 8 hours, and 30 minutes, 25 seconds with an inaccuracy of 0.300 seconds:

```
21-08:30:25.000I000.300
```

The following example shows a negative relative time of 20.2 seconds with an unspecified inaccuracy (default).

```
-20.2
```

The following example shows a relative time of 10 minutes, 15.1 seconds with an inaccuracy of 4 seconds:

```
10:15.1I4
```

Notice that a relative time does not use the calendar date fields, because these fields concern absolute time. A positive relative time is unsigned; a negative relative time is preceded by a **-** (minus) sign. A relative time is often subtracted from, or added to, another relative or absolute time. Relative times that DTS uses internally are opaque binary timestamps. The DTS API offers several routines that can be used to calculate new times using relative binary timestamps.

### Representing Periods of Time

A given duration of a period of time can be represented by a data element of variable length that uses the syntax shown in Figure 73 on page 373.

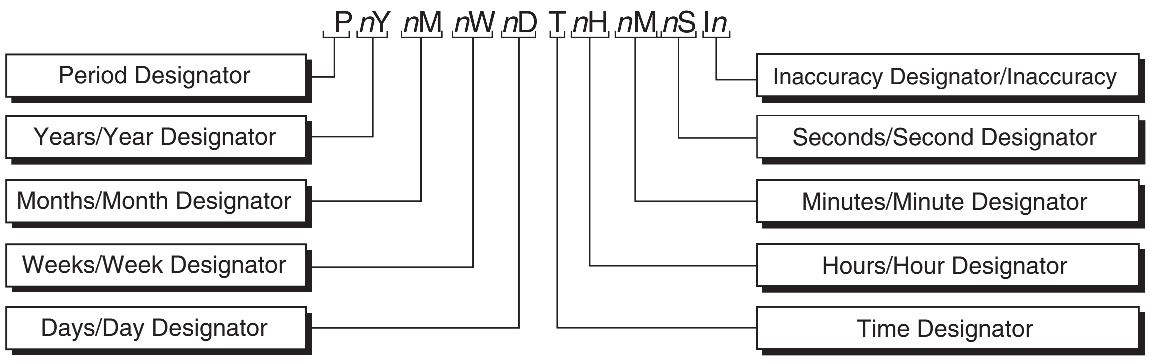


*Figure 73. Time Period Data Element Syntax*

The data element contains the following parts:

- The designator **P** precedes the part that includes the calendar components, including the following:

  **Y** The number of years followed by the designator

  **M** The number of months followed by the designator

  **W** The number of weeks followed by the designator

  **D** The number of days followed by the designator

- The designator **T** precedes the part that includes the time components, including the following:

  **H** The number of hours followed by the designator

  **M** The number of minutes followed by the designator

  **S** The number of seconds followed by the designator

- The designator **I** precedes the number of seconds of inaccuracy.

The following example represents a period of 1 year, 6 months, 15 days, 11 hours, 30 minutes, and 30 seconds and an unspecified inaccuracy:

```
P1Y6M15DT11H30M30S
```

The following example represents a period of 3 weeks and an inaccuracy of 4 seconds:

```
P3WI4
```

## Time Structures

DTS can convert between several types of binary time structures that are based on different base dates and time unit measurements. DTS uses UTC-based time structures, and can convert other types of time structures to its own presentation of UTC-based time. The DTS API routines are used to perform these conversions for applications on your system.

Table 20 on page 374 lists the absolute time structures that the DTS API uses to change binary times for applications.

*Table 20. Absolute Time Structures*

| Structure | Time Units | Base Date | Approximate Range |
|---|---|---|---|
| **utc** | 100-nanosecond | 15 October 1582 | A.D. 1 to A.D. 30,000 |
| **tm** | second | 1 January 1900 | A.D. 1 to A.D. 30,000 |
| **timespec** | nanosecond | 1 January 1970 | A.D. 1970 to A.D. 2038 |

Table 21 lists the relative time structures that the DTS API uses to change time structures for applications.

*Table 21. Relative Time Structures*

| Structure | Time Units | Approximate Range |
|---|---|---|
| **relutc** | 100-nanosecond | +/- 30,000 years |
| **tm** | second | +/- 30,000 years |
| **reltimespec** | nanosecond | +/- 68 years |

The remainder of this section explains the DTS time structures in detail.

## The utc Structure

Coordinated Universal Time (UTC) is useful for measuring time across local time zones and for avoiding the seasonal changes (summer time or daylight savings time) that can affect the local time. DTS uses 128-bit binary numbers to represent time values internally; throughout this book, these binary numbers representing time values are referred to as ***binary timestamps***. The DTS **utc** structure determines the ordering of the bits in a binary timestamp; all binary timestamps that are based on the **utc** structure contain the following information:

- The count of 100-nanosecond units since 00:00:00.00, 15 October 1582 (the date of the Gregorian reform to the Christian calendar)
- The count of 100-nanosecond units of inaccuracy applied to the preceding item
- The Time Differential Factor (TDF)
- The DTS version number
- The DST flag.

The binary timestamps that are derived from the DTS **utc** structure have an opaque format. This format is a cryptic character sequence that DTS uses and stores internally. The opaque binary timestamp is designed for use in programs, protocols, and databases.

**Note:** Applications use the opaque binary timestamps when storing time values or when passing them to DTS.

The API provides the necessary routines for converting between opaque binary timestamps and other strings that can be displayed and read by users.

## The tm Structure

The **tm** structure is based on the time in years, months, days, hours, minutes, and seconds since 00:00:00 GMT (Greenwich Mean Time), 1 January 1900. The **tm** structure is defined in the <**time.h**> header file.

The **tm** structure declaration is shown in Figure 74:

```
struct tm {
               int tm_sec;    /* Seconds (0 - 59)                  */
               int tm_min;    /* Minutes (0 - 59)                  */
               int tm_hour;   /* Hours (0 - 23)                    */
               int tm_mday;   /* Day of Month (1 - 31)             */
               int tm_mon;    /* Month of Year (0 - 11)            */
               int tm_year;   /* Year - 1900                       */
               int tm_wday;   /* Day of Week (Sunday = 0)          */
               int tm_yday;   /* Day of Year (0 - 364)             */
               int tm_isdst;  /* Nonzero if Daylight Savings Time  */
                              /*  is in effect                     */
          };
```

*Figure 74. tm Structure Declaration*

Not all of the **tm** structure fields are used for each routine that converts between **tm** structures and **utc** structures. (See the parameter descriptions that accompany the routines in *z/OS DCE Application Development Reference* for additional information about which fields are used for specific routines.)

## The timespec Structure

The **timespec** structure is normally used in combination with or in place of the **tm** structure to provide finer resolution for binary times. The **timespec** structure is similar to the **tm** structure, but the **timespec** structure specifies the number of seconds and nanoseconds since the base time of 00:00:00 GMT, 1 January 1970. You can find the structure in the **<dce/utc.h>**. header file.

The **timespec** structure declaration is shown in Figure 75:

```
struct timespec {

                 time_t tv_sec; /*  Seconds since 00:00:00 GMT,     */
                                /*   1 January 1970                        */
                 long tv_nsec;  /*  Additional nanoseconds since           */
                                /*   tv_sec                                */

                }         timespec_t;
```

*Figure 75. timespec Structure Declaration*

## The reltimespec Structure

The **reltimespec** structure represents relative time. You can find the **reltimespec** structure in the **<dce/utc.h>** header file.

The **reltimespec** structure declaration is shown in Figure 76:

```
struct reltimespec {

                    long tv_sec;   /*  Seconds of relative time      */
                    long tv_nsec;  /*  Additional nanoseconds of     */
                                   /*   relative time                */

                   }   reltimespec_t;
```

*Figure 76. reltimespec Structure Declaration*

## DTS API Header Files

The **time.h** and **<dce/utc.h>** header files contain the data structures, type definitions, and define statements that are referred to by the DTS API routines. The **<dce/utc.h>** header file includes **time.h**, and contains the **timespec**, **reltimespec**, and **utc** structures.

The **<dce/utc.h>** header file is located in **/usr/include/dce**.

## DTS API Routine Functions

Figure 77 on page 377 categorizes the DTS portable interface routines by function.

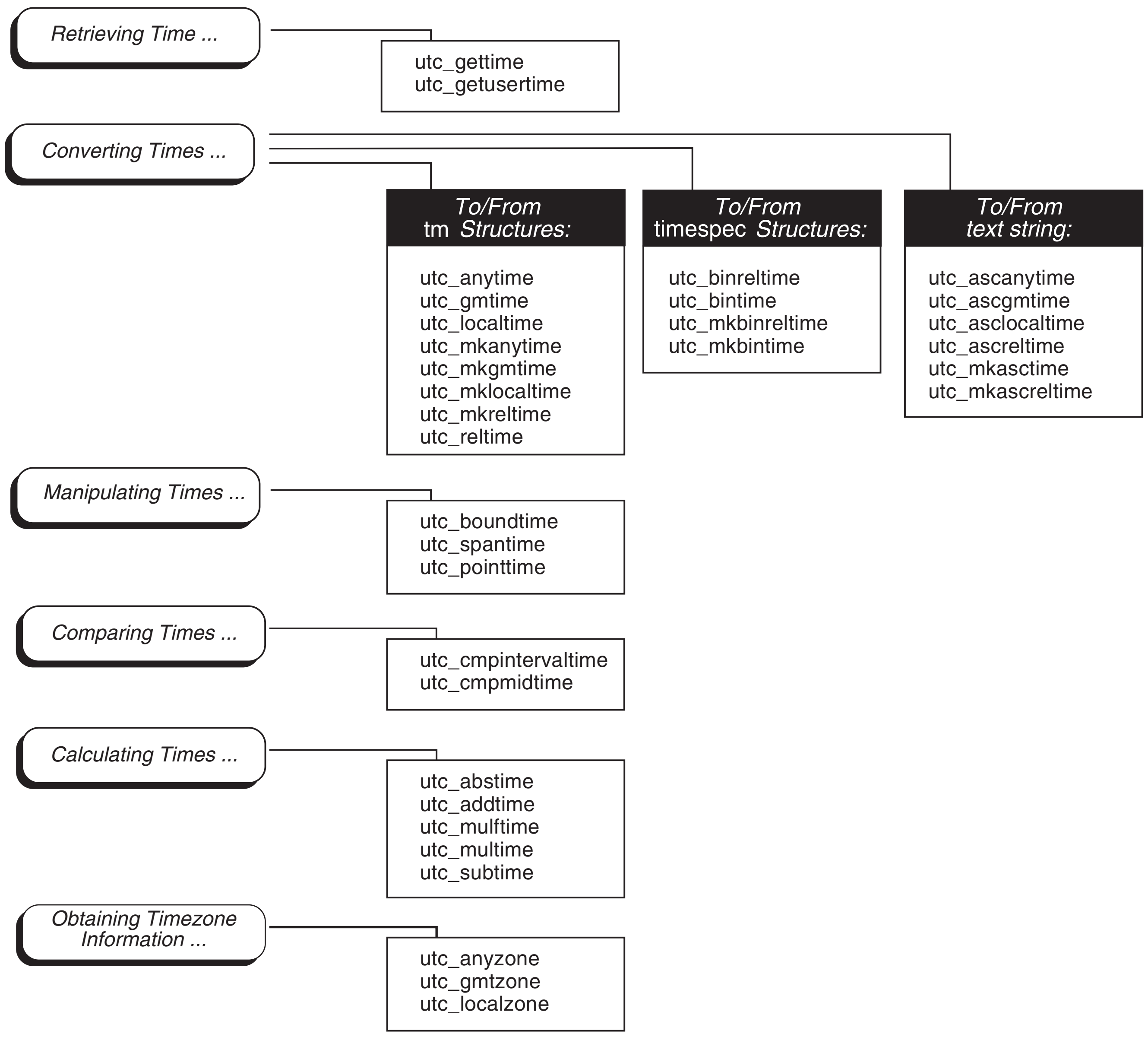


*Figure 77. DTS API Routines Shown by Functional Grouping*

An alphabetical listing of the DTS portable interface routines and a brief description of each one follows:

| | |
|---|---|
| **utc_abstime** | Computes the absolute value of a binary relative timestamp. |
| **utc_addtime** | Computes the sum of two binary timestamps; the timestamps can be two relative times or a relative time and an absolute time. |
| **utc_anytime** | Converts a binary timestamp into a **tm** structure, using the TDF information contained in the timestamp to determine the TDF returned with the **tm** structure. |
| **utc_anyzone** | Gets the time zone label and offset from GMT, using the TDF contained in the input **utc**. |
| **utc_ascanytime** | Converts a binary timestamp into a text string that represents an arbitrary time zone. |
| **utc_ascgmtime** | Converts a binary timestamp into a text string that expresses a GMT time. |

| | |
|---|---|
| **utc_asclocaltime** | Converts a binary timestamp to a text string that represents a local time. |
| **utc_ascreltime** | Converts a binary timestamp that expresses a relative time to its text string representation. |
| **utc_binreltime** | Converts a relative binary timestamp into two **timespec** structures that express relative time and inaccuracy. |
| **utc_bintime** | Converts a binary timestamp into a **timespec** structure. |
| **utc_boundtime** | Given two UTC times, one before and one after an event, returns a single UTC time whose inaccuracy includes the event. |
| **utc_cmpintervaltime** | Compares two binary timestamps or two relative binary timestamps. |
| **utc_cmpmidtime** | Compares two binary timestamps or two relative binary timestamps, ignoring inaccuracies. |
| **utc_gettime** | Returns the current system time and inaccuracy as an opaque binary timestamp. |
| **utc_getusertime** | Returns the time and process-specific TDF, rather than the system-specific TDF. |
| **utc_gmtime** | Converts a binary timestamp into a **tm** structure that expresses GMT or the equivalent UTC. |
| **utc_gmtzone** | Gets the time zone label, given **utc**. |
| **utc_localtime** | Converts a binary timestamp into a **tm** structure that expresses local time. |
| **utc_localzone** | Gets the time zone label and offset from GMT, given **utc**. |
| **utc_mkanytime** | Converts a **tm** structure and TDF (expressing the time in an arbitrary time zone) into a binary timestamp. |
| **utc_mkascreltime** | Converts a null-terminated character string, which represents a relative timestamp, to a binary timestamp. |
| **utc_mkasctime** | Converts a null-terminated character string, which represents an absolute timestamp, to a binary timestamp. |
| **utc_mkbinreltime** | Converts a **timespec** structure expressing a relative time to a binary timestamp. |
| **utc_mkbintime** | Converts a **timespec** structure into a binary timestamp. |
| **utc_mkgmtime** | Converts a **tm** structure that expresses GMT or UTC to a binary timestamp. |
| **utc_mklocaltime** | Converts a **tm** structure that expresses local time to a binary timestamp. |
| **utc_mkreltime** | Converts a **tm** structure that expresses relative time to a binary timestamp. |
| **utc_mulftime** | Multiplies a relative binary timestamp by a floating-point value. |
| **utc_multime** | Multiplies a relative binary timestamp by an integer factor. |
| **utc_pointtime** | Converts a binary timestamp to three binary timestamps that represent the earliest, most likely, and latest time. |
| **utc_reltime** | Converts a binary timestamp that expresses a relative time into a **tm** structure. |
| **utc_spantime** | Given two (possibly unordered) binary timestamps, returns a single UTC time interval whose inaccuracy spans the two input timestamps. |
| **utc_subtime** | Computes the difference between two binary timestamps that express either an absolute time and a relative time, two relative times, or two absolute times. |

# Chapter 21. Time-Provider Interface

This chapter describes the Time-Provider Interface (TPI) for DCE Distributed Time Service software. It provides a brief overview of the TPI, explains how to use external time-providers with DTS, and describes the data structures and message protocols that make up the TPI.

Coordinated Universal Time (UTC) is widely used and is disseminated throughout the world by various standards organizations. Several manufacturers supply devices that can acquire UTC time values through radio, satellite, or telephone. These devices can then provide standardized time values to computer systems. Normally, one device is connected to a computer system; the device runs a process that interprets signals and translates them to time values, which can either be displayed or be provided to the server process running on the connected system.

To synchronize its system clock with UTC using an external time-provider device, a DTS server needs a software interface to the device to periodically obtain UTC. In effect, this interface serves as an intermediary between the DTS server and external time-provider processes. The DTS server requires the interface to obtain UTC time values and to determine the associated inaccuracy of each value. The interface between the DTS server process and the time-provider process is called the Time-Provider Interface (TPI).

The remainder of this chapter describes the TPI and its attendant processes in detail. The following section describes the control flow between the DTS server process, the TPI, and the Time-Provider process.

## General TPI Control Flow

When you use a time-provider with a system running DTS, the external time-provider is an independent process that communicates with a DTS server process through remote procedure calls (RPCs). A ***remote procedure call*** is a synchronous request and response between a main calling program and a procedure running in another process. RPC applications are based on the client/server model. In this context, the following processes act as the client and server components in the RPC-based application:

- The DTS daemon is the client.
- The Time-Provider process (TP process) is the server.

Both the RPC-client (DTS daemon) and the server (TP process) must be running on the same system.

Applications running on RPC communicate through an interface that is well known to both the client and the server. The RPC interface consists of a set of procedures, data types, and constants that describe how a client can call a routine running on the server. The server offers the interface to the clients through the Interface Definition Language (IDL) file.

The IDL file defines the syntax for an operation, including the following:

- The name of the operation
- The data type of the value that the operation returns (if any)
- The order and data types of the operation's parameters (if any).

The TP process offers two procedures that DTS calls to obtain time values. These procedures are **ContactProvider** and **ServerRequestProviderTime**.

At each system synchronization, DTS makes the initial remote procedure call (**ContactProvider**) to a TP process that is assumed to be running on the same node.

If the TP process is active, the RPC call returns the following arguments:

- A successful communication status message
- A control message that DTS uses for further processing.

If the TP process is not active, the RPC call either returns a communication status failure or a time out occurs. DTS then synchronizes with other servers instead of with the external time provider.

If the initial call (**ContactProvider**) is successful, DTS makes a second call (**ServerRequestProviderTime**) to retrieve the timestamps from the external time provider. The control message sent by the TP process in the first RPC call specifies the length of time DTS waits for the RPC call to be completed. The TP process returns the following parameters in the **ServerRequestProviderTime** procedure call:

- A communication status message
- A time structure that contains timestamps collected from the external time-provider. (DTS then uses these timestamps to complete its synchronization.)

Figure 78 illustrates the RPC calling sequence between DTS and the TP process. Solid black lines represent the path followed by input parameters; gray lines represent the path followed by output parameters and return values.

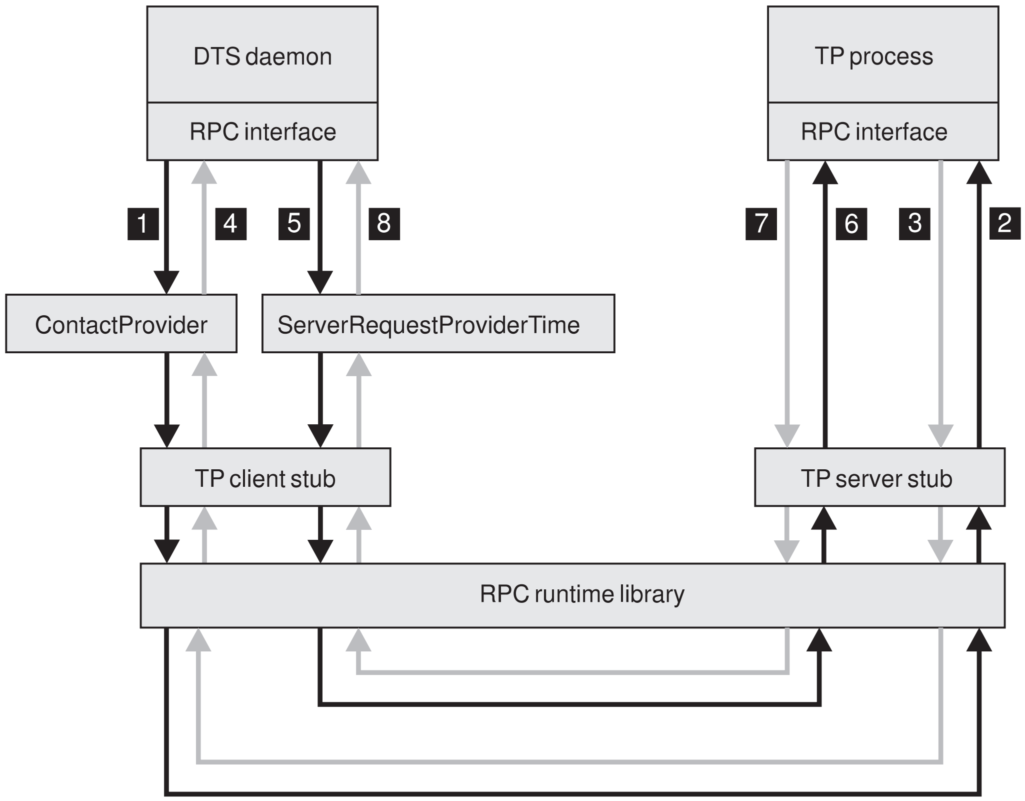


*Figure 78. DTS Time-Provider RPC Calling Sequence*

The following steps describe the process shown in Figure 78:

**1** At synchronization time, DTS calls the **ContactProvider** remote procedure. Input parameters are passed to the TP client stub, sent to the RPC runtime library, and then passed to the TP server stub.

**2** The TP process receives the call and runs the **ContactProvider** procedure.

**3** The procedure ends and returns the results through the TP server stub, the RPC runtime library, and the TP client stub.

**4** The procedure ends in the DTS call, where the returned parameters are examined.

**5** DTS then calls the **ServerRequestProviderTime** remote procedure. Input parameters are passed to the TP client stub, sent to the RPC runtime library, and then passed to the TP server stub.

**6** The TP process receives the call and runs the **ServerRequestProviderTime** procedure.

**7** The procedure ends and returns the results through the TP server stub, the RPC runtime library, and the TP client stub.

**8** The DTS RPC ends and the timestamps are returned as an output parameter. DTS then synchronizes using the timestamps returned by the external time provider.

The following section describes the remote procedures that are exported by the TP process during the previous sequence.

## ContactProvider Procedure

**ContactProvider** is the first routine called by DTS. The routine is called to verify that the TP process is running and to obtain a control message that DTS uses for subsequent communication with the TP process and for synchronization after it receives the timestamps. The parameters passed in the **ContactProvider** procedure call consist of the following elements:

Binding Handle
: An input parameter that establishes the relationship between DTS and the TP process. A binding handle enables the client (DTS) to recognize and find a server (the TP process) that offers the same interface.

Control Message
: An output parameter that contains information used by DTS for subsequent processing. The control message consists of the following elements:

  *TPstatus* One of the following values:

  - **K_TPI_SUCCESS**
  - **K_TPI_FAILURE**

  *nextPoll* A time value that tells DTS when to contact the TP process again. For example, once a day through dial-up, radio, or satellite.

  *timeout* A value that tells DTS how long to wait for a response from the TP process.

  *noClockSet* A value that specifies whether or not DTS is allowed to alter the system clock. If *noClockSet* is specified as 0x01 (True), DTS does not adjust or set the clock during the current synchronization. This option is useful for systems whose system clock is known to be accurate (such as systems equipped with special hardware) or systems that are managed by some other time service (such as Network Time Protocol (NTP)), but which still want to function as a DTS server.

| | |
|---|---|
| Communication Status | An output parameter that contains a status code returned by the DCE RPC runtime library. The status **rpc_s_ok** is returned if the TP process is successfully contacted. |

## ServerRequestProviderTime Procedure

After the TP process is successfully contacted, DTS makes the **ServerRequestProviderTime** procedure call to obtain the timestamps from the external time-provider. The parameters passed in the **ServerRequestProviderTime** procedure call consist of the following elements:

Binding Handle
: An input parameter that establishes the relationship between DTS and the TP process. A binding handle enables the client (DTS) to recognize and find a server (the TP process) that offers the same interface.

Time Response Message
: An output parameter that contains a TP process status value (**K_TPI_SUCCESS** or **K_TPI_FAILURE**), a count of the timestamps that are returned, and the timestamps obtained from the external time-provider. The timestamp count is an integer in the range **K_MIN_TIMESTAMPS** to **K_MAX_TIMESTAMPS**. Each timestamp consists of three **utc** time values:

  - The system clock time immediately before the TP process polls the external time source. (The TP process normally obtains the time from the **utc_gettime()** DTS API routine.)
  - The time value returned to the TP process by the external time source.
  - The system clock time immediately after the external time source is read. (The TP process obtains the time from the **utc_gettime()** DTS API routine.)

Communication Status
: An output parameter that contains a status code returned by the DCE RPC runtime library. The status **rpc_s_ok** is returned if the TP process is successfully contacted.

## Time-Provider Process IDL File

An RPC can only work if an interface definition that clearly defines operation signatures exists. Operation signatures define the syntax for an operation, including its name and parameters (input and output) that are passed as part of the procedure call. The TP process interface exports the two operation signatures that have been previously explained. You can find this interface in the file **dce/dtsprovider.idl**. When you build the TP process application, you must compile this file using the Interface Definition Language (IDL) compiler. The IDL compiler creates three files:

- **dtsprovider.h** (header file)
- **dtsprovider_sstub.c** (server stub file)
- **dtsprovider_cstub.c** (client stub file)

The Time-Provider program must be compiled along with the server stub code and then linked together. The TP program must also include the stub-generated file, **dtsprovider.h**. The sample code contained in Figure 79 on page 383 shows the structure of this interface.

```
/*
 *              Time Service Provider Interface
 *
 * This interface is defined through the Network Interface
 * Definition Language (NIDL).
 */


[uuid (bfca1238-628a-11c9-a073-08002b0dea7a),
    version(1)
]

interface time_provider
{

 import "dce/nbase.idl";
 import "dce/utctypes.idl";

/*
 * Minimum and Maximum number of times to read time source at each
 * synchronization
 */

const long K_MIN_TIMESTAMPS   = 1;
const long K_MAX_TIMESTAMPS   = 6;

/*
 * Message status field return values
 */
const long K_TPI_FAILURE        = 0;
const long K_TPI_SUCCESS        = 1;

/*
 * This structure contains one reading of the TP wrapped in the
 * timestamps of the local clock.
 */

typedef struct TimeResponseType
{
   utc_t beforeTime;            /* local clk just before getting UTC */
   utc_t TPtime;                /* source UTC; inacc also supplied   */
   utc_t afterTime;             /* local clk just after getting UTC  */
} TimeResponseType;
```

*Figure 79 (Part 1 of 4). Time Service Provider Interface*

```
/*
 * Time-provider control message.  This structure is returned in
 * response to a time service request.  The status field returns TP
 * success or failure.  The nextPoll gives the client the time at
 * which to poll the TP next. The timeout value tells the client how
 * long to wait for a time response from the TP.  The noClockSet will
 * tell the client whether or not it is allowed to alter the system
 * clock after a synchronization with the TP.
 */

typedef struct TPctlMsg
{
    unsigned long       status;
    unsigned long       nextPoll;
    unsigned long       timeout;
    unsigned long       noClockSet;
} TPctlMsg;


/* TP timestamp message.  The actual time-provider synchronization
 * data.  The status is the result of the operation (success or
 * failure).  The timeStampCount parameter returns the number of
 * timestamps being returned in this message.  The timeStampList is
 * the set of timestamps being returned from the TP.
 */

typedef struct TPtimeMsg
{
    unsigned long       status;
    unsigned long       timeStampCount;
    TimeResponseType    timeStampList[K_MAX_TIMESTAMPS];

} TPtimeMsg;
```

*Figure 79 (Part 2 of 4). Time Service Provider Interface*

```
/*
 * The Time-Provider Interface structures are described here.
 * There are two types of response messages from the TP:
 * control message and data message.
 *
 *        <<<< TPI CONTROL MESSAGE >>>>
 *
 * 31                                          0
 * +--------------------------------------------+
 * |            Time-Provider Status            |
 * +--------------------------------------------+
 * |            Next Poll Delta                 |
 * +--------------------------------------------+
 * |            Message Time Out                |
 * +--------------------------------------------+
 * |            NoSet Flag                      |
 * +--------------------------------------------+
 *
 *
 *     <<<< a single timestamp >>>>
 *
 * 128                                         0
 * +--------------------------------------------+
 * |            Before Time                     |
 * +--------------------------------------------+
 * |            TP Time                         |
 * +--------------------------------------------+
 * |            After Time                      |
 * +--------------------------------------------+
 *
 *
 *     <<<< TPI DATA MESSAGE >>>>
 *
 * 31                                          0
 * +--------------------------------------------+
 * |            Time-Provider Status            |
 * +--------------------------------------------+
 * |            Timestamp Count                 |
 * +--------------------------------------------+
 * |                                            |
 * |            <timestamp one>                 |
 * |                                            |
 * +--------------------------------------------+
 * |                  .                         |
 * |                  .                         |
 * |                  .                         |
 * |                  .                         |
 * |                  .                         |
 * +--------------------------------------------+
 * |                                            |
 * |            <timestamp K_MAX_TIMESTAMPS>    |
 * |                                            |
 * +--------------------------------------------+
 *
```

*Figure 79 (Part 3 of 4). Time Service Provider Interface*

```
 */
/*
 * The RPC-based Time-Provider Program (TPP) interfaces are defined
 * here. These calls are run by a Time Service daemon running as
 * a server (in this case it makes an RPC client call to the TPP server).
 */

/*
 * CONTACT_PROVIDER
 *
 * Send initial contact message to the TPP.  The TPP server
 * responds with a control message.
 */

void ContactProvider
        (
        [in]    handle_t        bind_h,
        [out]   TPctlMsg        *ctrlRespMsg,
        [out]   error_status_t  *comStatus
        );

/*
 * SERVER_REQUEST_PROVIDER_TIME
 *
 * The client sends a request to the TPP for times.
 * The TPP server responds with an array of timestamps
 * obtained by querying the Time-Provider hardware that it polls.
 */

void ServerRequestProviderTime
        (
        [in]    handle_t        bind_h,
        [out]   TPtimeMsg       *timesRspMsg,
        [out]   error_status_t  *comStatus
        );

}
```

*Figure 79 (Part 4 of 4). Time Service Provider Interface*

## Initializing the Time-Provider Process

Initializing the RPC-based TP process prepares it to receive remote procedure calls from a DTS daemon requesting the timestamps. The following steps are involved:

1. Include the header file (**provider.h**) that is created by compiling **dce/provider.idl**, which contains the interface definition.
2. Register the interface with the DCE RPC runtime.
3. Select one or more protocol sequences that are compatible with both the interface and the runtime library. It is recommended that the TP process application selects all protocol sequences available on the system. Available protocol sequences are obtained by calling an RPC API routine, described in the example that follows. This ensures that transport-independence is maintained in RPC applications.
4. Register the TP process with the endpoint mapper service (**dced**) running on the system. This makes the TP process available to the DTS daemon.

5. Obtain the name of the machine's principal and then register an authentication service to use with authenticated RPCs coming from the DTS daemon. Note that DTS and the Time Provider program are presumed to be running in an authenticated environment.
6. Listen for remote procedure calls.

The example in Figure 80 illustrates these steps, including the sequence of calls needed.

```
/*
 * Register the TP server interface with the RPC runtime.
 * The interface specification time_provider_v1_0_ifspec
 * is obtained from the generated header file dtsprovider.h
 * The entry point vector is normally defined at the top of
 * the TP source program similar to this:
 *
 *    globaldef time_provider_v1_0_epv_t time_provider_epv =
 *    {
 *       ContactProvider,
 *       ServerRequestProviderTime
 *    };
 *
 */
rpc_server_register_if (time_provider_v1_0_s_ifspec,
                        NULL,
                        (rpc_mgr_epv_t) &time_provider_epv,
                        &RPCstatus);

/*
 * This call tells the DCE RPC runtime to listen for remote
 * procedure calls using all supported protocol sequences.
 * To listen for a specific protocol sequence, use the
 * rpc_server_use_protreq call.
 */

rpc_server_use_all_protseqs (max_calls,
                             &RPCstatus);

/*
 * This routine is called to obtain a vector of binding handles
 * that were established with registration of protocol sequences.
 */
rpc_server_inq_bindings (&bind_vector,
                         &RPCstatus);

/*
 * This routine adds the address information of the binding
 * handle for the TP server to the endpoint mapper database.
 */

rpc_ep_register (time_provider_v1_0_s_ifspec,
                 bind_vector,
                 NULL,
                 "Time Provider",
                 &RPCstatus);
```

*Figure 80 (Part 1 of 2). Initializing the Time-Provider Process*

```
/*
 * Obtain the name of the machine’s principal and register an
 * authentication service to use for authenticated remote procedure
 * calls coming from the time service daemon.
 */

dce_cf_prin_name_from_host (NULL,
                            &machinePrincipalName,
                            &status);

rpc_server_register_auth_info (machinePrincipalName,
                               rpc_c_authn_dce_private,
                               NULL,
                               NULL,
                               &RPCstatus);

/*
 * This routine is called to listen for remote procedure calls
 * send by the DTS client.  The possible RPC calls coming from
 * the DTS client are ContactProvider and ServerRequestProviderTime.
 */

rpc_server_listen (max_calls,
                   &RPCstatus);
```

*Figure 80 (Part 2 of 2). Initializing the Time-Provider Process*

## Time-Provider Algorithm

The time-provider algorithm assumes that the two RPCs will come in the following order: **ContactProvider** followed by **ServerRequestProviderTime**. The algorithm to create a generic time-provider follows:

1. Initialize the TP process, as previously described. Listen for RPC calls.
2. If the **ContactProvider** procedure is started, perform the following steps:
   a. Initialize the control message to the appropriate values (status value to **K_TPI_SUCCESS**; *nextPoll*, *timeout*, and *noClockSet* to valid integer values).
   b. Set the communication status output parameter to **rpc_s_ok**.
   c. Return from the procedure call. (The DCE RPC runtime returns the values to DTS.)
3. If the **ServerRequestProviderTime** procedure is run, perform the following steps:
   a. Initialize the timestamp count to the appropriate number.
   b. Use the **utc_gettime()** DTS API routine to read the system time.
   c. Poll the external time source and read a UTC value. Use the **utc_gmtime()** routine to convert the UTC time value to a binary timestamp.
   d. Use the **utc_gettime()** routine to read the system time.
   e. Repeat steps b, c, and d the number of times specified by the values of **K_MIN_TIMESTAMPS** and **K_MAX_TIMESTAMPS**.
   f. If steps b, c, or d return erroneous data, initialize the TP process status field (*TPstatus*) of the data message to **K_TPI_FAILURE**; otherwise, initialize the data message timestamps.
   g. Set the communication status output parameter to **rpc_s_ok**.
   h. Return from the procedure call. (The DCE RPC runtime sends the values back to DTS.)

4. The TP process continues listening for RPC calls.

## DTS Synchronization Algorithm

DTS performs the following steps to synchronize with an external time-provider:

1. At startup time, create the binding handle for the Time-Provider interface. The binding handle is obtained from the list of available protocol sequences on the system.
2. At synchronization time, make the remote procedure call **ContactProvider**, assuming that a TP process is running on the system. If the procedure call fails, examine the RPC communication status, checking the availability of the server. If the server is unavailable, synchronize with peer servers; otherwise continue.
3. Wait for the procedure call to return the control message in the output parameter. If it does not return within the specified LAN time out interval, synchronize with peer servers. Otherwise go to step 4.
4. If the procedure call returned successfully (communication status is **rpc_s_ok**), read the data in the control message.
5. Make the RPC **ServerRequestProviderTime** to obtain the timestamps from the external time-provider. If the procedure does not return within the elapsed time specified by the control message (*timeout*), synchronize with peer servers. Schedule the next synchronization based on the applicable DTS management parameters, ignoring *nextPoll*.
6. If the procedure returns successfully, verify that the TP process status is **K_TPI_SUCCESS**. Otherwise synchronize with peer servers and schedule the next synchronization.
7. Extract the timestamps from the data message and synchronize using the timestamps.
8. Schedule the next synchronization time by adding the value of *nextPoll* seconds to the current time. At the next synchronization, go to step 2.

**Note:** Application developers do not have to perform these steps; DTS performs these steps internally during synchronization with an external time-provider.

## Running the Time-Provider Process

Both the TP process and the DTS daemon must run on the same system. The TP process must be started up under the login context of the machine's principal, which has authenticated user privileges. The DTS daemon and the TP process are started independently. However, before starting the TP process, ensure that the endpoint mapper daemon (**dced**) is running on the system. If it is not running, start it. The TP process can always exit without affecting the DTS daemon. DTS dynamically reestablishes communications with the TP process when it wants to synchronize.

## Running a User-Written Time Provider Program

To run a time provider that you have written in the DCEKERN address space, do the following:

1. Change the name of the time provider program name in the **/opt/dcelocal/etc/euvpdcf** file to your time provider program name.
2. Change the parameters to your time provider program if required.
3. Alter the DCEKERN cataloged procedure so that the your time provider program can be run.

Before running your time provider program in the DCEKERN address space, ensure that it is fully tested and of production quality. Any problems with your time provider may impact the operation of the DCEKERN address space. Note that you can run your time provider program outside the DCEKERN address space the same way as any other DCE application. It must run using the authorized id to machine principal.

**Note:** The status of user built time provider programs will be reported as *unknown* when using the TSO MODIFY command to check the status.

## Sources of Additional Information

Refer to the following for additional information:

- Look in the **examples/dts** file for examples of time-provider programs that you can use with several different types of external time-provider devices.
- See *z/OS DCE Administration Guide* for commercial sources of external time providers.
- See *z/OS DCE Application Development Reference* for detailed information about the RPC API and DTS API routines.

# Chapter 22. DTS API Routines Programming Example

This chapter contains a C programming example showing a practical application of the DTS API programming routines. The program performs the following actions:

- Prompts the user to enter two sets of time coordinates corresponding to the timestamps of two *events*.
- Stores those coordinates in a **tm** structure.
- Converts the **tm** structure to a **utc** structure.
- Prints out the **utc** structure in ISO text format.
- Determines which event occurred first.
- Determines if Event 1 may have caused Event 2 by comparing the intervals.

```
#include <time.h>   /* time data structures                              */
#include <dce/utc.h>    /* utc structure definitions                     */

void ReadTime();
void PrintTime();

/*
 * This program requests user input about events, then prints out
 * information about those events.
 */

main()
{
    struct utc event1,event2;
    enum utc_cmptype relation;

    /*
     * Read in the two events.
     */

    ReadTime(&event1);
    ReadTime(&event2);

    /*
     * Print out the two events.
     */

    printf("The first event is : ");
    PrintTime(&event1);
    printf("\nThe second event is : ");
    PrintTime(&event2);
    printf("\n");

    /*
     * Determine which event occurred first.
     */

    if (utc_cmpmidtime(&relation,&event1,&event2))
        exit(1);

    switch( relation )
    {
        case utc_lessThan:
        printf("comparing midpoints: Event1 < Event2\n");
        break;
        case utc_greaterThan:
        printf("comparing midpoints: Event1 > Event2\n");
        break;
        case utc_equalTo:
        printf("comparing midpoints: Event1 == Event2\n");
        break;
        default:
        exit(1);
        break;
    }
```

*Figure 81 (Part 1 of 3). An Example of a DTS Program*

```
    /*
     * Could Event 1 have caused Event 2?  Compare the intervals.
     */

    if (utc_cmpintervaltime(&relation,&event1,&event2))
        exit(1);

    switch( relation )
    {
        case utc_lessThan:
        printf("comparing intervals: Event1 < Event2\n");
        break;
        case utc_greaterThan:
        printf("comparing intervals: Event1 > Event2\n");
        break;
        case utc_equalTo:
        printf("comparing intervals: Event1 == Event2\n");
        break;
        case utc_indeterminate:
        printf("comparing intervals: Event1 ? Event2\n");
        default:
        exit(1);
        break;
    }

}

/*
 * Print out a utc structure in ISO text format.
 */

void PrintTime(utcTime)
struct utc *utcTime;
{

    char   string[50];

    /*
     * Break up the time string.
     */

    if (utc_ascgmtime(string,      /* Out: Converted time    */
                      50,          /* In:  String length     */
                      utcTime))    /* In:  Time to convert   */
        exit(1);
    printf("%s\n",string);

}

/*
 * Prompt the user to enter time coordinates.  Store the coordinates
 * in a tm structure and then convert the tm structure to a utc structure.
 */
```

*Figure 81 (Part 2 of 3). An Example of a DTS Program*

```
void ReadTime(utcTime)
struct utc *utcTime;
{
struct tm tmTime,tmInacc;

    (void)memset((void *)&tmTime,  0, sizeof(tmTime));
    (void)memset((void *)&tmInacc, 0, sizeof(tmInacc));
    (void)printf("Year? ");
    (void)scanf("%d",&tmTime.tm_year);
    tmTime.tm_year -= 1900;
    (void)printf("Month? ");
    (void)scanf("%d",&tmTime.tm_mon);
    tmTime.tm_mon -= 1;
    (void)printf("Day? ");
    (void)scanf("%d",&tmTime.tm_mday);
    (void)printf("Hour? ");
    (void)scanf("%d",&tmTime.tm_hour);
    (void)printf("Minute? ");
    (void)scanf("%d",&tmTime.tm_min);
    (void)printf("Inacc Secs? ");
    (void)scanf("%d",&tmInacc.tm_sec);

    if (utc_mkanytime(utcTime,
                      &tmTime,
                      (long)0,
                      &tmInacc,
                      (long)0,
                      (long)0))
        exit(1);

}
```

*Figure 81 (Part 3 of 3). An Example of a DTS Program*

# Part 5. Using the DCE Security APIs

This part of the book provides details on the DCE Security services and facilities, and describes the main Security interfaces. Walkthroughs of Authentication and Authorization are presented to enhance your understanding of the DCE security concepts. You are shown how to use the DCE Security APIs, GSSAPI credentials, and the DCE Audit Service and Audit API functions.

# Chapter 23. Overview of Security

This chapter provides a brief overview of the two security services available in DCE:

- DCE Security Service
- Generic Security Service (GSS)

Refer to *z/OS DCE Application Development Reference* for detailed information on the application program interfaces (APIs) discussed in this section.

Support for Kerberos V5R1 is provided. This support is in the form of an API to allow for cross-platform interoperability with non-DCE systems. This support assumes that you are already familiar with Kerberos and that you are porting an *existing* application that uses Kerberos to z/OS. For *new* applications, use the DCE API instead.

All details of the Kerberos support are found in other DCE books. For more information, see:

- *z/OS DCE Configuring and Getting Started* for Kerberos configuration files
- *z/OS DCE Administration Guide* for Kerberos environment variables
- *z/OS DCE Application Development Reference* for Kerberos-related APIs

## Purpose and Organization of Security Section

The discussions in the Security section explain the major features of DCE Security, so you can decide what you require to ensure that your DCE application is sufficiently secure. Many security features are built into DCE, so in many cases you need to do little or nothing to secure your DCE application. Furthermore, you do not need to understand all the details of DCE Security to use it effectively.

Following the overview of DCE Security in this chapter are two chapters that contain conceptual discussions of ***authentication*** and ***authorization***. The remaining chapters in this part discuss:

- GSSAPI Credentials
- Extended Privilege Attribute API
- Registry API
- Extended Attribute API
- Login API
- Key Management API
- Access Control List API
- ID Map API
- DCE Audit Service
- Using the Audit API functions
- Password Management API
- RACF-DCE Interoperability

## About Authenticated RPC

Perhaps the most important API for ensuring secure DCE applications is the Authenticated RPC facility. It enables distributed applications to participate in authenticated network communications. Applications using the Authenticated RPC routines may use the ***authentication protocol*** and the ***authorization protocol***, and set various protocol-independent ***protection levels*** for communicating with remote ***principals***. *Principals* may refer to users, servers and computers.

Using Authenticated RPC is explained in “Authenticated RPC Routines” on page 94. This section contains conceptual information that assists in understanding the authentication and authorization protocols that Authenticated RPC routines use. For a greater understanding of Authenticated RPC, read Chapter 24, “Authentication” on page 413 and Chapter 25, “Authorization” on page 435.

## About the Generic Security Service API

The Generic Security Service (GSS) provides an alternate way of providing DCE security to distributed applications that handle network communications by themselves. With GSSAPI, you can include established applications in DCE, and ensure the security and integrity of the applications and their data. In peer-to-peer communications, the application that establishes the secure connection is the *context initiator* or simply *initiator*. The context initiator is like a DCE RPC client. The application that accepts the secure connection is the *context acceptor* or simply *acceptor*. The context acceptor is like a DCE RPC server.

The GSS available with DCE includes two sets of routines:

- Standard GSSAPI routines. These routines have the prefix **gss_**.
- OSF DCE extensions to the GSSAPI routines. These are additional routines that enable an application to use DCE security services. These routines have the prefix **gssdce_**.

The chapters that follow provide information about how the GSSAPI routines use the authentication and authorization protocols. Chapter 26, “GSSAPI Credentials” on page 447 provides information about GSS credentials, which are used to establish an application's identity in DCE.

## What Authentication and Authorization Mean

There are two questions that DCE Security can answer for a principal about another principal with which it may want to communicate:

- Is this principal really what it says it is?
- Does it have the right to do what it wants to do?

Depending on the answers to these questions, a security-sensitive principal takes different courses of action with respect to a principal with which it communicates.

To authenticate a principal means to verify that the principal represents its true identity. To authorize a principal means to grant ***permission*** for the principal to perform an operation. The concepts of authentication and authorization are intertwined. A principal’s authorization is explicitly linked to its identity. Additionally, there is the possibility that authorization data concerning an authenticated principal can be falsified. This raises the question:

*Should the authorization data concerning this principal be believed?*

DCE Security can answer this question for a concerned principal whether the principal acts as the identity of an application server or a client.

The specific mechanisms by which authentication and authorization are performed are called authentication and authorization protocols. DCE Security supports at least one of each. However, RPC documentation (see “Authenticated RPC Routines” on page 94) refers to authentication and authorization protocols as services.

The GSSAPI combines authentication and authorization under a single security mechanism type. The security mechanism provides applications a choice of either authenticated Kerberos security or authenticated PAC authorization under DCE Security.

## RACF Authorization using RACF-DCE Interoperability

A DCE application server can use DCE security services for controlling access to resources owned by the application server. As an alternative to this method, when developing applications you may want to use the authorization and auditing capabilities provided by Resource Access Control Facility (RACF) for the server portion of a client-server application that resides on z/OS. With z/OS DCE, this is possible using the information created when z/OS users are enrolled in the RACF-DCE interoperability feature. For information on enrolling in and using RACF-DCE interoperability, see *z/OS DCE Administration Guide*. For more information on using RACF-DCE interoperability, see Chapter 37, “RACF-DCE Interoperability Application Programming Interfaces” on page 535.

## Authentication, Authorization, and Data Protection in Brief

When one principal talks to another in a distributed computing environment, there is a risk that communications between the two will provide a means for compromising the security of one or the other. For example, a client may attack a server, or a server may set a trap for clients. An attack is most likely to succeed if the malevolent principal can convince its victim that it is something other than what it really is (an attacker), and/or that it possesses authorization that it does not really have. A counterfeit identity and/or authorization data grants an attacker access that it presumably would not otherwise have, and so provides an opportunity for the attacker to do damage.

One way an attacker might obtain counterfeit credentials is to intercept network transmissions between a client and a server, and then attempt to decipher (and perhaps modify) the transmitted data. If the attacker is able to intercept *and decipher* a principal's authentication or authorization information, it can later use this data to masquerade as an authentic principal with proper authorization.

DCE Security protects against these kinds of attacks. It contains features that enable principals to

- Detect whether data they receive has been modified in transit
- Be reasonably certain that an attacker will be unable to decipher any authentication and authorization data it may succeed in intercepting.

DCE Security gives DCE principals confidence that the identity and authorization of principals they communicate with are authentic.

Figure 82 on page 403 is an extremely condensed and highly stylized representation of the essentials of DCE Security in terms of the DCE Shared-Secret authentication protocol and the DCE Authorization protocol. Unless we note otherwise, assume that discussions in this Part of this guide refer to these two protocols, used in conjunction with one another.

Following is a description of the events depicted in the illustration:

1. Principal A (which could be an attacker masquerading as Principal A) requests authentication of its identity from the Authentication Service. This request is encrypted using several keys, one of which is a key derived from the password supplied by Principal A. A copy of Principal A's key also exists in the Registry database, having been stored there when the principal's account was created (or when the password was changed.) It is thus available to the Authentication Service.

   The Authentication Service then obtains the Registry's copy of Principal A's key and uses it to decrypt Principal A's authentication request. If the decryption succeeds, the keys are the same; Principal A is therefore authenticated and the Authentication Service replies with information that enables Principal A to ask the Privilege Service to authenticate its privilege attributes. (Privilege attributes are data used in making authorization decisions; they consist of the principal's name and group memberships.) If Principal A fails to get authenticated privilege attributes (sometimes referred to as *credentials*), it may simply assert its privilege attributes to Principal B.

2. Principal A now makes a request to Principal B to perform some operation that requires the **c** permission to object **d**, and presents its certified privilege attributes. Principal B may grant or deny **c** access to **d** after examining the Access Control List (ACL) that protects object **d** (an ACL associates the privilege attributes of principals with permissions to an object). If **c** is one of the permissions listed in the ACL entry that specifies the permission set that may be granted to Principal A, then Principal A is allowed to perform the operation; if the **c** permission is not listed in that entry, A is denied access.

   Had the Authentication Service been unable to decrypt the principal's authentication request, the principal would have been unauthenticated, and as a consequence, unable to acquire certified privilege attributes from the Privilege Service. In that case, Principal A might have simply asserted its privilege attributes to B; that is, claimed them for itself, without the benefit of having the Privilege Service certify this data as being genuine. Had Principal A then presented asserted privilege attributes to Principal B, then B might have denied the requested permission or granted it, depending on whether B grants permissions to unauthenticated principals, and whether **c** is among the permissions that B grants to such principals.

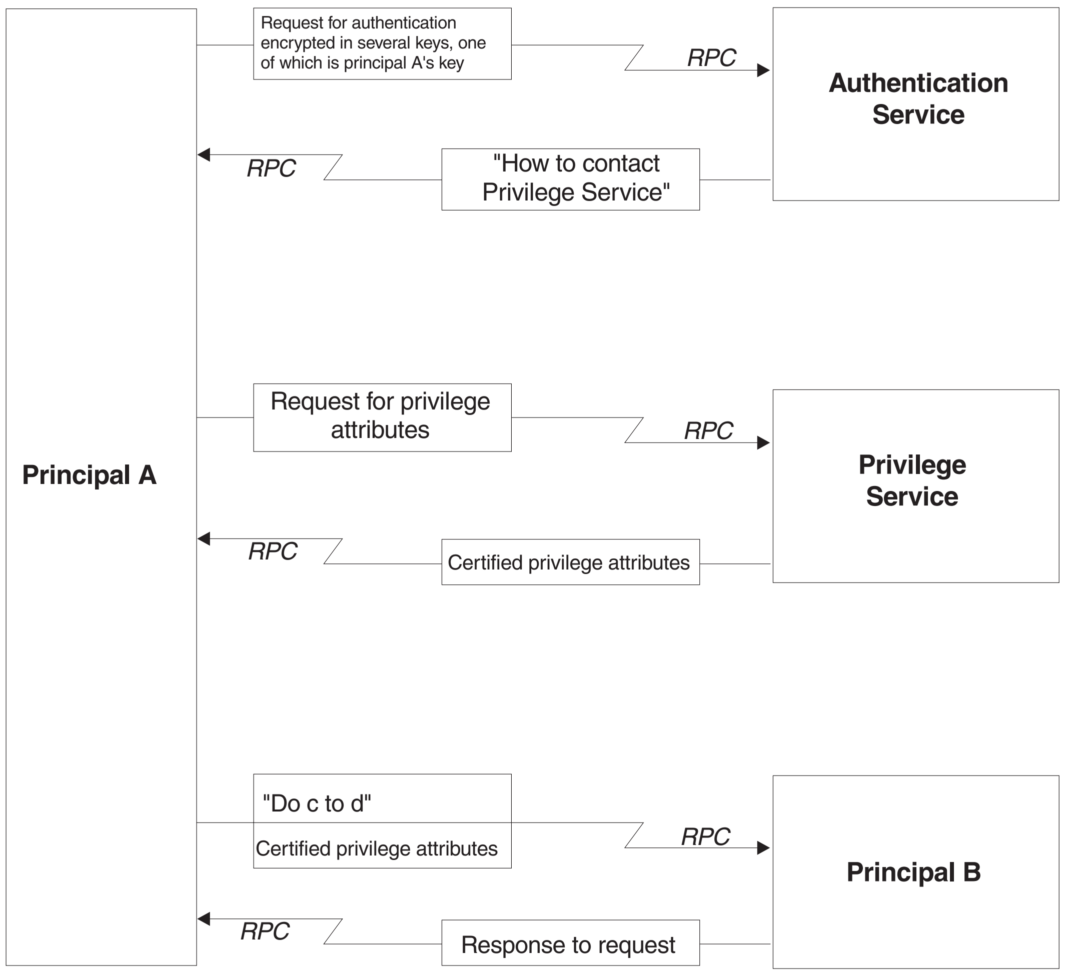

Request for authentication encrypted in several keys, one of which is principal A's key
RPC
Authentication Service
RPC
"How to contact Privilege Service"
Request for privilege attributes
RPC
Principal A
Privilege Service
RPC
Certified privilege attributes
"Do c to d"
Certified privilege attributes
RPC
Principal B
RPC
Response to request

*Figure 82. Shared-Secret Authentication and DCE Authorization in Brief*

If Principals A and B are especially sensitive to security concerns, they may request that RPC data be checked for integrity to establish whether it has been changed in transit. They can also request encryption to ensure that the data is unintelligible to any party other than Principals A and B.

## Summary of DCE Security Services and Facilities

The DCE Security component consists of services and facilities.

### Security Services

The services are:

- The ***Registry Service***, which maintains a database of principals, groups, organizations, accounts, and administrative policies.
- The ***Authentication Service***, which verifies the identity of a principal and issues ***tickets*** used by the principal to access remote services (a ticket is data about a principal that is presented to the principal providing the service).
- The ***Privilege Service***, which certifies a principal's privilege attributes (that is, its name and group memberships, which are represented as UUIDs). Permissions, as embodied by ACLs, are enforced

by the Privilege Service, but are defined and maintained by the Access Control List facility, described below.

The three security services are available from a single daemon, the Security server.

## Security Facilities

The security facilities are:

- The ***Login facility***, which enables a principal to establish its network identity.
- The ***Extended Registry Attribute (ERA) facility***, which extends the Registry database to maintain attribute types and instances.
- The ***Extended Privilege Attribute (EPA) facility***, which provides access to the information in extended privilege attribute certificates (EPACs).
- The ***Access Control List (ACL) facility***, which enables a principal's access to an object to be determined by a comparison of the principal's privilege attributes to the object's permissions.
- The ***Key Management facility***, which enables noninteractive principals (most frequently, servers) to manage their secret keys.
- The ***ID Map facility***, which maps cell-relative principal names to global principal names, and global principal names to cell-relative principal names. This facility is used in connection with the transmission of information about principals that are members of different DCE cells.
- The ***Password Management facility***, which enables principals' passwords to be generated, and to be subjected to strength checks beyond those defined in DCE standard policy.

**Registry Service Interfaces:** Command line interfaces to the Registry Service are described in the z/OS DCE administration documentation. Following is a summary:

**dcecp** or **rgy_edit** Edits Registry database entries.

The API to the Registry Service consists of calls with the **sec_rgy** prefix. Because this is the same interface that the Registry Service user and administration tools call, few applications use it, unless they replace some or all of the functionality of the default Registry tools.

**Authentication Service Interfaces:** Following is a summary of the command line interfaces to the Authentication Service when the default authentication protocol is in effect (the default protocol is DCE Shared-Secret, which is based on the ***Kerberos*** Version 5 network authentication system). These interfaces are described in the z/OS DCE administration documentation:

| | |
|---|---|
| **kinit** | Obtains a login session's ticket(s) to remote services. |
| **klist** | Lists a login session's tickets to remote services. |
| **kdestroy** | Destroys a login session's tickets to remote services. |

There are two sets of security APIs that distributed applications are most likely to call to use the Authentication Service:

- Authenticated RPC facility
- GSSAPI

Although an application that uses GSSAPI may not make explicit calls to RPC routines, the GSSAPI implementation itself uses DCE RPC to communicate with the DCE registry.

**Privilege Service Interfaces:** There are neither command line interfaces nor application program interfaces to the Privilege Service. The Login facility and authenticated RPC or GSSAPI encapsulate interactions between a principal and the Privilege Service.

## Interfaces to the Login Facility

Command line interfaces to the Login facility consists of the following tool (described in the z/OS DCE administration documentation):

**dce_login** Enables an interactive principal to log into DCE, but does not change the principal's local identity.

The API to the Login facility consists of calls with the **sec_login** prefix. This API enables application processes to assume their network identities.

## Interfaces to the Extended Registry Attribute Facility

The user interface to the ERA facility consists of **dcecp** subcommands that allow users to modify the registry schema to create and maintain attribute types and to create and maintain instances of those types.

The API to the ERA facility consists of calls that are prefixed with **sec_rgy_attr**.

## Interfaces to the Extended Privilege Attribute Facility

There are no user interfaces to the Extended Privilege Attribute facility. The API to this facility consists of calls that are prefixed with **sec_cred**. These routines extract data from EPACs.

## Interfaces to the Key Management Facility

It can be important for a DCE server application to have a network identity that is distinct from the user principal identity that runs it, or the host on which it runs. As the server is a noninteractive principal, it requires a way in which to pass its *key* to the Authentication Service, other than by typing in a *password* as interactive principals do. The recommended method is to store server keys in a locally protected key table. With the Key Management facility, noninteractive principals such as servers can manage their secret keys.

The command line interface to the Key Management facility consists of a few **dcecp** or **rgy_edit** subcommands. These subcommands enable an administrator to change or delete the secret key of a noninteractive principal if such a key is compromised. These subcommands call the Key Management API, which consists of several calls with the prefix **sec_key**.

## Interfaces to the ID Map Facility

There are no command line interfaces to the ID Map facility. The API to this facility consists of calls with the prefix **sec_id**. These routines map a global principal or group name into a cell name and a cell-relative principal or group name, and generate a global principal or group name from a cell name and a cell-relative principal or group name. This API also converts the internal (UUID) representation of a name to a human-readable string and back again.

## Interfaces to the Access Control List Facility

The command line interface to the ***Access Control List facility*** is the **dcecp** tool. This tool edits an object's ACL whose entries specify the permissions to the object that may be granted to principals possessing specified privilege attributes. (The **dcecp** tool is described in *z/OS DCE Administration Guide*.)

The ACL API consists of routines that are prefixed **sec_acl**. This is the same API that **dcecp** calls, so an ACL editor or browser that replaces **dcecp** calls this API. A different case is an application server that needs to store and retrieve application-specific access-control information for its clients. Such an application must put into effect its own ACL manager using the subset of the ACL routines that are prefixed **dce_acl_mgr** as a guide. The ACL manager must also export the ACL manager interface. (Refer to Chapter 32, "The Access Control List Application Program Interfaces" on page 505 for more information on ACL managers.) The **dcecp** command is the client-side user interface to an ACL Manager. You write the ACL Manager as part of your DCE server application.

## Interfaces to the Password Management Facility

The user interface to the Password Management facility is provided by subcommands to the **rgy_edit** and **dcecp** commands. These subcommands enforce password management policy for principals, and enable them to request generated passwords. See the *z/OS DCE Administration Guide* for information on the **rgy_edit** and **dcecp** commands and instructions on how to create and change principal passwords.

The API to the Password Management facility consists of routines that are prefixed with **sec_pwd_mgmt**. See Chapter 36, "The Password Management Application Programming Interfaces" on page 531 and the *z/OS DCE Application Development Reference* for information on these routines.

## Interfaces to RACF-DCE Interoperability

The administrator interface to enroll z/OS users in RACF-DCE interoperability is provided by the z/OS DCE **mvsexpt** and **mvsimpt** utilities. See *z/OS DCE Administration Guide* for information on these utilities. It is strongly recommended that you use these utilities to establish and maintain the association between DCE principals and RACF user IDs. The utilities manage DCE information contained in the RACF user profile and in the RACF DCEUUIDS class, which provides the cross linking between a user's DCE identity and the corresponding RACF identity.

RACF also provides extensions to RACF administrative commands for the RACF-DCE interoperability, as well as ISPF panel support that provides a subset of this administrative capability. See *z/OS SecureWay Security Server RACF Security Administrator's Guide*, SA22-7683, for more information on the RACF support.

The APIs to this information are a combination of DCE APIs, z/OS UNIX System Services callable services, and C library function calls. The DCE APIs whose names start with **rpc_binding** and **sec_cred** are used to obtain the DCE principal's identity from DCE security credentials. The z/OS UNIX System Services callable services are:

- **auth_check_resource_np()**, which is used to check access to a RACF-protected resource
- **convert_id_np()**, which is used to obtain a z/OS user ID for a corresponding DCE UUID.

The IBM C/C++ library provides function calls for these z/OS DCE callable services. The C function call names that provide C language bindings for these callable services are:

- **__check_resource_auth_np()**
- **__convert_id_np()**

See *z/OS C/C++ Run-Time Library Reference*, SA22-7821, for a discussion of these C function library calls. Information on the z/OS UNIX System Services callable services can be found in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*, SA22-7803.

z/OS users enrolled in RACF-DCE interoperability and single sign-on to DCE can store their DCE passwords in RACF by using the **storepw** command. The password is also stored in the DCE registry if the **-r** option is specified on the command. See *z/OS DCE Command Reference* and *z/OS DCE User's Guide* for more information on the **storepw** command.

## Relationships Between the Security Service and DCE Applications

Figure 83 is a schematic illustration of the relationships among the interfaces to the DCE Security Service and the relationship of Security interfaces to DCE applications.

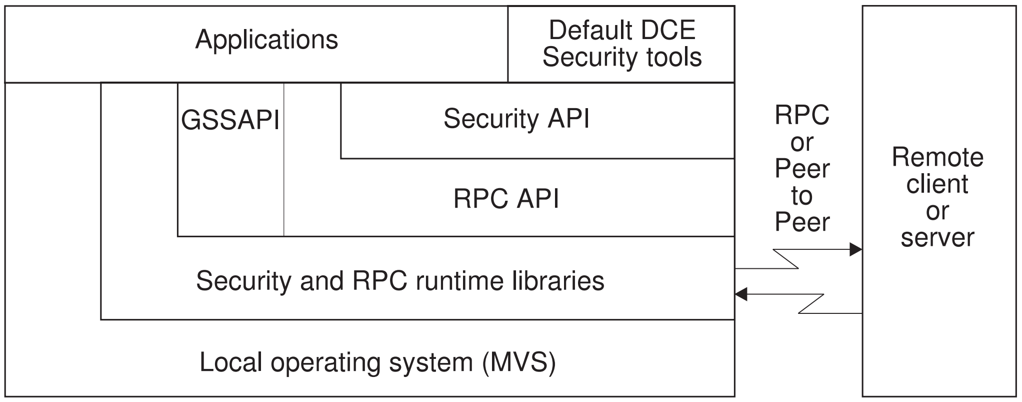

Applications
Default DCE Security tools
GSSAPI
Security API
RPC API
Security and RPC runtime libraries
Local operating system (MVS)
RPC or Peer to Peer
Remote client or server

*Figure 83. DCE Security and the DCE Application Environment*

## DTS, the Cell Namespace, and Security

This section discusses the dependencies of DCE Security on the Distributed Time Service (DTS) and the relationship between the Security namespace and the Cell Directory Service (CDS) namespace. For information on how DCE components (such as CDS) use features of DCE Security, refer to the documentation on the component of interest. For example, refer to *z/OS DCE Administration Guide* section about CDS.

### Time and Security

The Security Service depends on the DTS to maintain a relatively close synchronization of network clocks. When network clocks become skewed — that is, different machines have clocks that differ by seconds or minutes — you may regard unexpired tickets to services as not valid or expired tickets as valid. Excessive skewing probably inconveniences users rather than introducing opportunities for security breaches. Administrative intervention is required. See *z/OS DCE Administration Guide* for information on how to monitor and manage timer skew.

## The Cell Namespace and the Security Namespace

The Registry database maintains three Security namespaces: the principal, group, and organization (PGO) namespaces. These namespaces are distinct from the cell namespace maintained by the Cell Directory Service. Security names take the following form:

*/.../cell_name/pgo_name*

Whereas CDS names take the following form:

*/.../cell_name/mount_point/object_name*

Because the Security namespace is rooted in the CDS namespace, Security names have equivalent CDS names. Thus, for example, an entry for a principal in the Registry database has the following form in the Security namespace:

*/.../cell_name/principal_name*

It has the following form in the CDS namespace:

*/.../cell_name*/sec/principal/*principal_name*

The Security mount-point name (*sec* as shown in the preceding syntax) is determined when the DCE is configured. Therefore, the name may differ at individual sites.

There is no ambiguity about the Security namespace to which a name refers because Security names are always supplied in contexts that identify the namespace in question. For example, logging into DCE requires a principal name to be supplied.

However, an ACL is an object that is indirectly referred to by the name of the object it protects. Protected objects are not always Security objects, and therefore may be registered *only* in the CDS namespace. ACL management interfaces always take CDS names rather than Security names as input, whether it is the ACL of a Security object (such as a Registry database entry) that is being read or changed.

## Using DCE Three Different Ways

There are three mutually exclusive ways to use DCE on z/OS, with three different Dynamic Load Libraries (DLLs) :

- You can use the full range of DCE services by specifying -DMVS when compiling your application. You then link your application with **libdce.a** and specify the EUVPDLL.*x* side file. If you want to also use the DCE thread model, you must specify -D_DCE_THREADS when compiling your application modules. Your application is then bound to the EUVPDLL DLL.
- You can use just the DCE security services by specifying -DMVS -D_DLL_LIBDCE when compiling your application. You then link your application with **libsdce.a** and specify the EUVSDLL.*x* side file. Your application is then bound to the EUVSDLL DLL. If you use this method, you cannot use the DCE thread model nor can you use RPC, Time, or Directory services. In order to use the EUVSDLL DLL, you must run the DCE Security Server on the same system as the application program. More information on this method of using DCE is provided in “Using the DCE Security Services DLL” on page 409.
- You can use the Kerberos services by specifying -DMVS when compiling your application modules. You then link your application with **libkrb5.a** and specify the EUVFDLL.*x* side file. Your application is then bound to the EUVFDLL.

## Using the DCE Security Services DLL

DCE on z/OS provides a subset of the DCE application programming interfaces in a stand-alone DLL. This DLL does not use any of the DCE platform, RPC, Directory, or Time services. Program execution is considerably faster because the DCE security server is accessed using z/OS cross-memory services instead of using RPC and TCP/IP. In order to use the DCE Security Services DLL, you must run the DCE security server on the same system as the application program. In a parallel sysplex environment, this means you need to run the security server on each system in the sysplex.

An uninitialized replica security server is unable to provide any services until it has been initialized. As a result, local DCE security services are not available until the replica has been initialized. Applications attempting to use local DCE security services receive an error status of **sec_rgy_server_unavailable** until this process has been completed.

Applications receive an error status of **sec_rgy_server_unavailable** if an attempt is made to access a remote DCE security server in a context which does not support remote access. For example, the **sec_login_setup_identity()** routine only supports principals in the local cell and the **sec_login_import_context()** routine only imports contexts created for principals in the local cell. However, service tickets and privilege ticket-granting tickets can be obtained for remote cells. This means that an application using the DCE Security Services DLL can access services in a remote cell (but the application must provide its own network transport mechanism since RPC is not available).

Only the following environment variables are supported by the DCE Security Services DLL:

- _EUV_ENVAR_FILE
- _EUV_EXC_ABEND_DUMPS
- _EUV_HOME
- _EUV_SEC_KRB5CCNAME_FILE

For more information on these environment variables, see the *z/OS DCE Administration Guide*.

These DCE APIs are supported by the DCE Security Services DLL:

- dce_error_inq_text
- gss_accept_sec_context
- gss_acquire_cred - Only credentials for the local cell may be acquired
- gss_add_cred - Only credentials for the local cell may be added
- gss_add_oid_set_member
- gss_canonicalize_name
- gss_compare_name
- gss_context_time
- gss_create_empty_oid_set
- gss_delete_sec_context
- gss_display_name
- gss_display_status
- gss_duplicate_name
- gss_krb5_get_ccache
- gss_krb5_get_tkt_flags
- gss_get_mic
- gss_import_name
- gss_indicate_mechs
- gss_init_sec_context
- gss_inquire_context
- gss_inquire_cred
- gss_inquire_cred_by_mech

- gss_inquire_mechs_for_name
- gss_inquire_names_for_mech
- gss_oid_to_str
- gss_process_context_token
- gss_release_buffer
- gss_release_cred
- gss_release_name
- gss_release_oid
- gss_release_oid_set
- gss_str_to_oid
- gss_test_oid_set_member
- gss_unwrap
- gss_verify_mic
- gss_wrap
- gss_wrap_size_limit
- gssdce_cred_to_login_context
- gssdce_export_cred
- gssdce_extract_creds_from_sec_context
- gssdce_extract_PAC_from_sec_context
- gssdce_extract_PAC_from_cred
- gssdce_import_cred
- gssdce_login_context_to_cred
- gssdce_register_acceptor_identity
- gssdce_set_cred_context_ownership
- rpc_string_free
- sec_cred_free_cursor
- sec_cred_get_authz_session_info
- sec_cred_get_delegate
- sec_cred_get_delegation_type
- sec_cred_get_initiator
- sec_cred_get_pa_data
- sec_cred_initialize_cursor
- sec_cred_is_authenticated
- sec_id_gen_group
  - Specify NULL for the registry handle
  - The group permissions must allow unauthenticated read access
- sec_id_gen_name
  - Specify NULL for the registry handle
  - The principal permissions must allow unauthenticated read access
- sec_id_parse_group
  - Specify NULL for the registry handle
  - The group permissions must allow unauthenticated read access
- sec_id_parse_name
  - Specify NULL for the registry handle
  - The principal permissions must allow unauthenticated read access
- sec_login_become_delegate
- sec_login_become_impersonator
- sec_login_become_initiator
- sec_login_context_token_owner
- sec_login_create_context_token
- sec_login_cred_get_delegate
- sec_login_cred_get_expiration
- sec_login_cred_get_initiator
- sec_login_cred_init_cursor

- sec_login_delete_context_token
- sec_login_disable_delegation
- sec_login_expand_context_token
- sec_login_export_context
- sec_login_export_context_data
- sec_login_free_net_info
- sec_login_get_current_context
- sec_login_get_expiration
- sec_login_import_context – The imported context must be for a principal in the local cell
- sec_login_import_context_data – The imported context must be for a principal in the local cell
- sec_login_inquire_net_info
- sec_login_purge_context
- sec_login_release_context
- sec_login_refresh_identity
- sec_login_set_context
- sec_login_setup_identity – The principal must be in the local cell
- sec_login_valid_from_keytable
- sec_login_valid_from_system
- sec_login_validate_identity
- uuid_compare
- uuid_create
- uuid_create_nil
- uuid_equal
- uuid_from_string
- uuid_hash
- uuid_is_nil
- uuid_to_string

For more information on these APIs, see the *z/OS DCE Application Development Reference*.

# Chapter 24. Authentication

This chapter explains DCE Shared-Secret authentication concepts. The DCE Shared-Secret authentication protocol is the only authentication protocol supported by the Authenticated RPC facility and the GSSAPI. This protocol is the chief subject of this chapter.

For specific information about using the Authenticated RPC routines, refer to "Authenticated RPC Routines" on page 94.

## Background Concepts

This section presents a few background concepts that are useful for understanding the discussions of authentication in this chapter:

- Principals, which are the subjects of authentication.
- The cell, which is the environment where authentication occurs.
- The Shared-Secret Authentication protocol, which is the mechanism used for authentication when specified by applications through the Authenticated RPC facility.
- Protection levels, which are the various degrees that RPC data may be protected.
- Data encryption algorithms, which are the mechanisms that the Security Server and client runtimes use to encrypt and decrypt data exchanged between principals.

## Principals

The term *principal* was defined in Chapter 23, "Overview of Security" on page 399. A more precise definition of *principal* is: an entity that assumes it can communicate securely with another entity.

In DCE Security, principals are represented as entries in the Registry database. DCE principals include:

- Users, who are also referred to as *interactive principals*
- Instances of DCE servers
- Instances of application servers
- Computers in a DCE cell
- Authentication Service surrogates.

Each Registry database entry representing a principal contains the name of the principal and a secret key that the principal shares with the Authentication Service. (It is the secret key that enables a principal to solve the *puzzle* from the Authentication Service.) For a user, the secret key is derived from the user's password. To establish its identity as a principal, a noninteractive principal, such as a server or computer, must store its secret key in a data file or hardware device, or rely on a system administrator to enter it.

The Security Server itself comprises three principals that correspond to the three services it provides: Registry, Privilege, and Authentication.

**Note:** The Authentication Service principal is an exception because it does not share its key with any other principal. Authentication Service surrogates are also exceptions as they are not autonomous participants in authenticated communications, like other kinds of principals. Authentication surrogates resemble aliases for the Authentication Services of cells. Refer to "Intercell Authentication" on page 433 for more information on these subjects.

In the theory of Shared-Secret Authentication (and some other authentication protocols), all principals are untrusted, except for the Authentication Service itself. Therefore, a security-sensitive application authenticates all principals with which it communicates. However, because the Security Service integrates the Registry Service, the Privilege Service, and the Authentication Service (including its surrogates) as a single server process, any DCE application does not have to authenticate these principals.

## Cells and Realms

The *cell* is the basic unit of configuration and administration in DCE. In terms of Security, the set of principals that share a secret key with an instance of the Authentication Service is called a realm. In DCE, the cell and the realm correspond to the same entity. Therefore, each instance of a Security Server (not counting its replicas) defines a separate realm. The term *realm* may be more familiar to some readers than the term *cell*. A security realm is always configured to coincide with a corresponding CDS cell. z/OS DCE documentation refers to such a collective configuration of services as a *cell*.

## The Shared-Secret Authentication Protocol

With Authenticated RPC, you can specify the authentication protocol to use to authenticate principals. z/OS DCE does not support the Authentication protocols other than DCE Shared-Secret Authentication.

DCE Shared-Secret Authentication uses an extended version of the Kerberos Version 5 system as its authentication protocol. “A Walkthrough of the Shared Secret Authentication Protocol” on page 415 describes the protocol in general terms.

## Protection Levels

The protection level that an application sets using Authenticated RPC determines the level of encryption of network messages exchanged by principals. An application can set a protection level using either Authenticated RPC or GSSAPI.

**Authenticated RPC and Protection Levels:** The Authenticated RPC facility provides several levels of protection so that applications can control trade-offs between security and performance. Following is a summary of some of the protection levels that an application using Authenticated RPC can specify:

- Connect Level. Performs authentication only when a client and server establish a relationship.
- Call Level. Attaches a verifier to each client call and server response.
- Packet-integrity Level. Ensures that none of the data transferred between two principals has been changed in transit.
- Packet-privacy Level. Incorporates lesser protection levels and in addition encrypts all RPC argument values.

Refer to “Authenticated RPC Routines” on page 94 for complete information about protection levels.

**GSSAPI and Protection Levels:** Unlike secure RPC, where the client chooses a protection level that is then applied automatically to all data transferred in either direction, applications that use GSSAPI must protect data on a message-by-message basis. This allows an application the option of protecting only particularly sensitive messages, and avoids the overhead of security processing for other messages.

GSSAPI offers two distinct types of protection through the **gss_sign()** and **gss_verify()** routines and the **gss_seal()** and **gss_unseal()** routines, as follows:

- The **gss_sign()** routine creates a token containing a signature to protect the integrity of the message data. The token contains only the signature. The application must send both the token and the message to which it applies to the peer application for verification. The receiving application calls the **gss_verify()** routine to check the signature.
- The **gss_seal()** routine creates a token containing both a signature and the message data, and may optionally encrypt the data. Only the token need be sent to the peer application, which processes it using the **gss_unseal()** routine to verify the signature and extract the message data.

Three distinct signature algorithms are supported by the per-message protection routines. An algorithm may be requested by providing one of several constants to the **qop_request** parameter of either the **gss_sign()** or the **gss_seal()** routine. The constants are as follows:

| | |
|---|---|
| **GSSDCE_C_QOP_DES_MAC** | Conventional DES MAC. Slow but well understood. |
| **GSSDCE_C_QOP_DES_MD5** | DES MAC of an MD5 signature. Faster than QOP_DES_MAC. |
| **GSSDCE_C_QOP_MD5** | MD5 signature. Fastest supported signature algorithm. The default. |

## Data Encryption Mechanisms

The DCE security component of z/OS DCE uses the ***Data Encryption Standard*** (***DES***). and ***Common Data Masking Facility*** (***CDMF***) mechanisms for data privacy, and DES for principal authentication and data-integrity checking.

GSSAPI supports only DES encryption.

## A Walkthrough of the Shared Secret Authentication Protocol

This section presents a three-part walkthrough of the Shared-Secret authentication protocol:

- "A Walkthrough of User Authentication" on page 416 explains what happens when a user logs in using the default DCE login tool.
- "A Walkthrough of DCE Application Authentication" on page 425 explains what happens when the logged-in user runs an authenticated application.
- "A Walkthrough of DCE Application Authentication Using GSSAPI" on page 430 explains what happens when the logged-in user runs an application that uses GSSAPI.

Note that it is unnecessary to understand the Shared-Secret protocol in order to use it. It is described here for you to determine if it meets your security requirements. If you require more information read the next sections, otherwise, proceed to the next chapter.

The walkthrough is viewed primarily from the user and the associated application-client side. Schematic representations of events related to the protocol accompany the discussions. These illustrations do not show what literally happens when a user logs in and runs an authenticated application. They only provide a general understanding of the protocol.

In these illustrations, fill patterns represent encryption key values and encrypted data. When the key symbol appears in a box, it indicates a key is being passed as data. When the key symbol appears on a line, it indicates that encryption or decryption is taking place.

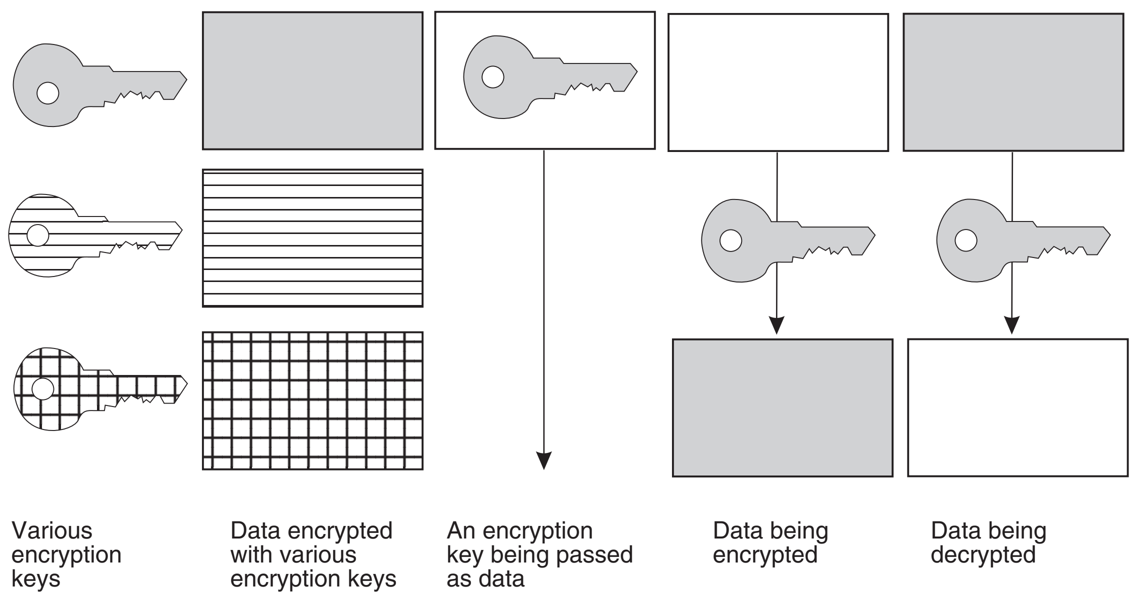

Various encryption keys
Data encrypted with various encryption keys
An encryption key being passed as data
Data being encrypted
Data being decrypted

*Figure 84. Representational Conventions Used in Authentication Walkthrough Illustrations*

## A Walkthrough of User Authentication

This section explains how DCE Security authenticates a user. DCE User authentication can be thought of as consisting of two successive procedures:

1. Acquisition by the Security Client of a Ticket-Granting Ticket (TGT) for the user
2. Acquisition by the Security Client of a Privilege-Ticket-Granting Ticket (PTGT) for the user

These procedures are described in the following two sections.

Note that this feature of DCE Security neither requires modification of DCE, nor the applications that run on it.

**How the Security Client Obtains a Ticket-Granting Ticket:** This section describes the acquisition, by the Security Client, of the user's Ticket-Granting Ticket. Acquisition of the user's TGT is the first of the two parts of DCE user authentication.

There are three protocols used by DCE Security clients and servers to perform this first part of the user-authentication process:

- The *third-party* protocol, which provides the highest level of security.
- The *timestamps* protocol, which is less secure .
- The OSF DCE 1.0 protocol, which is the least secure, and is provided solely to enable OSF DCE 1.1 security servers to process requests from pre-OSF DCE 1.1 clients.

The protocol used by the security client when it makes a login request to the Authentication Service is determined by:

- Pre-OSF DCE 1.1 clients always use the OSF DCE 1.0 protocol.

- OSF DCE 1.1 clients always use the third-party protocol, unless the host machine's session key, which the client uses to construct the request, is unavailable. It then uses the timestamps protocol.

The protocol used by the Authentication Service to respond to the client in is determined by:

- The protocol used by the client making the login request
- The value of a *pre_auth_req* ERA attached to the requesting principal

The Authentication Service always attempts to reply using the same protocol used by the client making the request, unless the value of the ERA forbids it to do so. (See the *z/OS DCE Administration Guide* for more detailed information on how security clients and the Authentication Service determine which protocol to use.)

For a general discussion of the security aspects of these protocols, and of security administration and security ERAs, see the *z/OS DCE Administration Guide*.

***The Third-Party Authentication Protocol:*** This section describes how the DCE Authentication Service uses the third-party authentication protocol to provide a user with a TGT. Refer to Figure 85 on page 418 as you read the following steps.

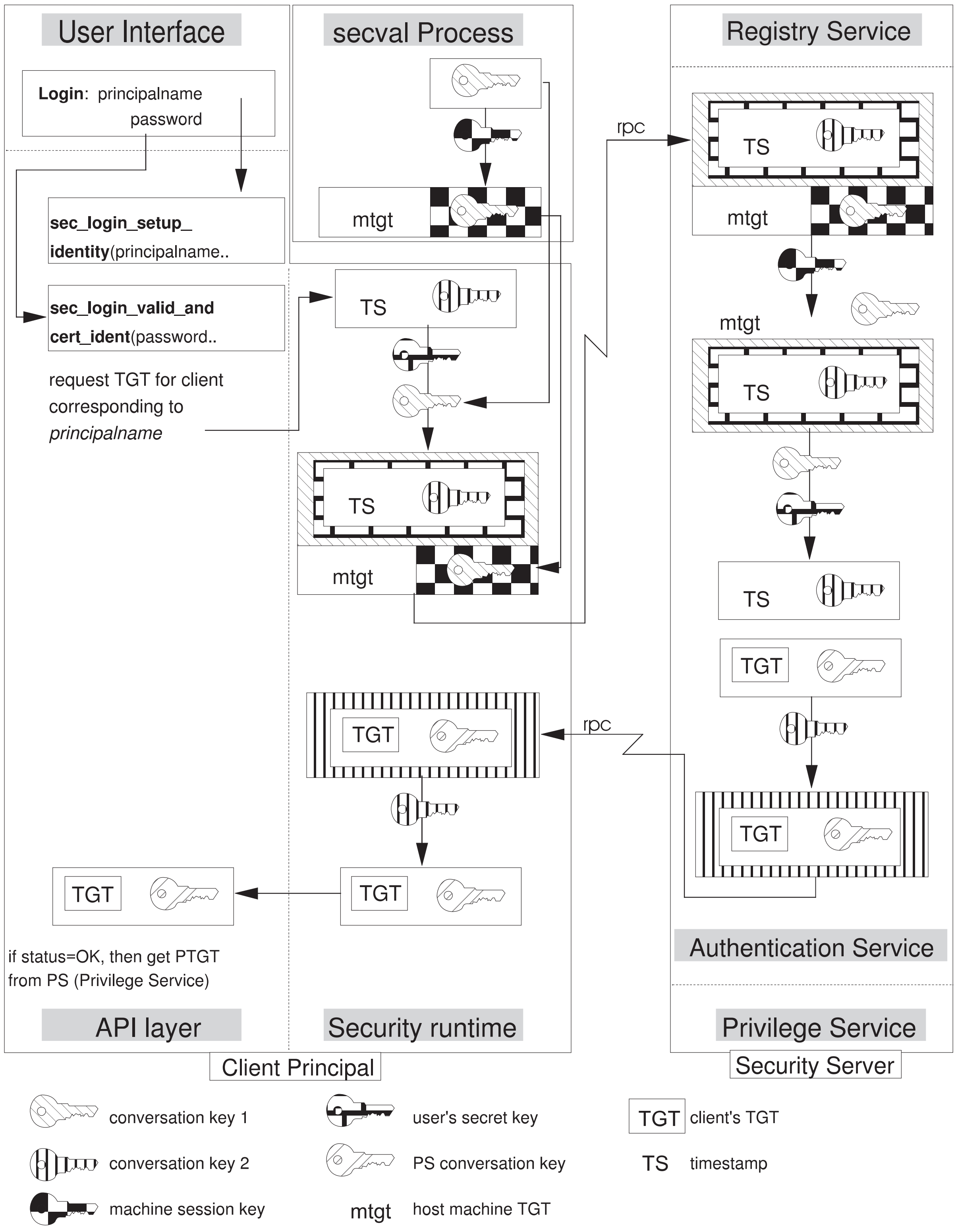


*Figure 85. Client Acquires Ticket-Granting Ticket Using Third-Party Protocol*

1. The user logs in, entering the correct user name. For examples of logging into DCE in batch or foreground mode, see the *z/OS DCE Administration Guide*. The login program calls **sec_login_setup_identity()**, which takes the user's principal name as one of its arguments, and **sec_login_valid_and_cert_ident()**, which has as one of its arguments the user's password. The

**sec_login_valid_and_cert_ident()** causes the Security runtime to request a Ticket-Granting Ticket (TGT) from the Authentication Service. A TGT enables a principal to be granted a ticket to a service of interest. In this case, it is the Privilege Service. The Security runtime performs the following steps to construct the TGT request:

   a. Requests, from the **secval** process, a string consisting of a random key (*conversation key 1*) encrypted using the machine session key (a copy of which also resides in the registry); then appends this string to the Machine Ticket-Granting Ticket (MTGT).

   b. Generates another random key (*conversation key 2*, which the Authentication Service will later use to encrypt the Ticket-Granting Ticket it returns to the client) and appends it to a timestamp string.

   c. Derives, from the password input by the user, the user's secret key, a copy of which also exists in the registry. Then encrypts the timestamp/conversation key 2 string twice: first using the user's secret key, and again using conversation key 1.

   d. Completes the authentication request by concatenating the string containing conversation key 1 (obtained from **secval** in Step 1a) with the doubly encrypted string containing the timestamp and conversation key 1.

2. The Security Runtime then forwards the TGT request to the Authentication Service.

3. The Authentication Service receives the request, and performs the following steps to verify the user and prepare the user's TGT:

   a. Identifies the machine principal by means of the machine Ticket-Granting Ticket, and obtains the machine session key for that machine from the Registry.

   b. Using the machine session key, decrypts the package containing conversation key 1.

      (Note that the ability of the authentication server to obtain the machine session key from the Registry and decrypt conversation key 1 verifies that it is the true Authentication Service server, and not an attacker.)

   c. Obtains, from the Registry, the user's secret key, and decrypts, using the user's secret key and conversation key 1, the package containing the timestamp and conversation key 2.

      If this decryption fails, the user's secret key that was used by the login program to encrypt the package differs from the one stored in the Registry, and therefore the password supplied to the login program by the user was incorrect. In this case, the user is not authenticated, and an error message is returned to the login program.

      If the decryption succeeds, and if the decrypted timestamp is within 5 minutes of the current time, the user has been verified (that is, the user knows the correct principal password), and the Authentication Service proceeds with preparation of the user's Ticket-Granting Ticket.

4. The Authentication Service then prepares the user's Ticket-Granting Ticket, encrypts it using conversation key 2 (which it received from the client Security Runtime) and returns the encrypted TGT to the client.

5. The client Security Runtime decrypts the reply from the Authentication Service using conversation key 2, obtains the user's Ticket-Granting Ticket, and it becomes part of the login context.

Note the following security safeguards inherent in the structure of this protocol:

- All network transmissions between Security client and Authentication Service are encrypted using strong random keys. All plaintext transmissions (which are vulnerable to off-line attacks) are double-encrypted, placing even off-line decryption attempts at the outer limits of practical possibility.
- The timestamp and conversation key 2 are encrypted using the user's secret key, which is derived from the user's password. This enables the Authentication Service to verify that the requesting client knows the user's password. (It does this by decrypting the package using the Registry's copy of the

user's secret key. If the decryption succeeds, the keys are the same. That is, they were derived from the same password.)

- The Authentication Service itself verifies whether the requesting client knows the user's password. It is therefore aware of, and can manage, persistent login failures for a given user, eliminating passive password-guessing attacks.
- The Authentication Service's reply is encrypted using conversation key 2, which was provided by the client. This verifies to the client that the Authentication Service is authentic, since, if it were not, it would not have been able to obtain from the Registry the machine session key and user's secret key it needed to decrypt conversation key 2.

These safeguards provide assurance to both server and client that the entity with which it is communicating is, in fact, what it claims to be.

Having acquired the user's Ticket-Granting Ticket, the login program then proceeds with Part 2 of the authentication procedure (described in "How the Client Obtains a Privilege-Ticket-Granting Ticket" on page 422).

***The Timestamps Authentication Protocol:*** This section describes how the DCE Authentication Service uses the timestamps authentication protocol to provide a user with a Ticket-Granting Ticket.

(Since the timestamps protocol is largely identical to the OSF DCE 1.0 protocol, which is fully explained in the next section, this section describes only the differences between the two.)

The timestamps protocol proceeds exactly as the OSF DCE 1.0 protocol described in the following section, with these additions:

- In Step 1, the client Security runtime sends to the Authentication Service, in addition to the user's name (UUID), a timestamp encrypted in the user's secret key.
- In Step 2, the Authentication Service, before preparing the user's TGT, verifies the user as follows:
  1. It decrypts the timestamp using the copy of the user's key it obtained from the Registry.
  2. If the decryption succeeds, and the timestamp is within 5 minutes of the current time, the user is verified, and the Authentication Service proceeds to prepare the TGT. If the decryption fails, or if the timestamp is not within 5 minutes of the current time, the Authentication Service rejects the login request.

With this protocol, the Authentication Service can verify:

- That the client login request is timely.
- That the requesting client knows the user's password.

It is therefore aware of, and can manage, persistent login failures for a given user, eliminating passive password-guessing attacks.

From this point, the timestamps protocol continues as the OSF DCE 1.0 protocol described in the next section, and then proceeds with Part 2 of the authentication procedure (described in "How the Client Obtains a Privilege-Ticket-Granting Ticket" on page 422).

***The OSF DCE 1.0 Authentication Protocol:*** This section explains how the DCE Authentication Service uses the OSF DCE 1.0 protocol to authenticate a user. This protocol exists in OSF DCE 1.1 solely to provide interoperability between OSF DCE 1.1 servers and pre-DCE1.1 clients; *only* pre-OSF DCE 1.1 clients transmit OSF DCE 1.0 login requests, and the Authentication Service returns OSF DCE 1.0 responses *only* to pre-DCE1.1 clients.

The OSF DCE 1.0 protocol lacks the security features described above for the third-party and timestamps protocols, and network transmissions using it are more susceptible to attacks on the user's TGT. You should keep this in mind when you are considering the inclusion of pre-OSF DCE 1.1 clients in your DCE1.1 cell.

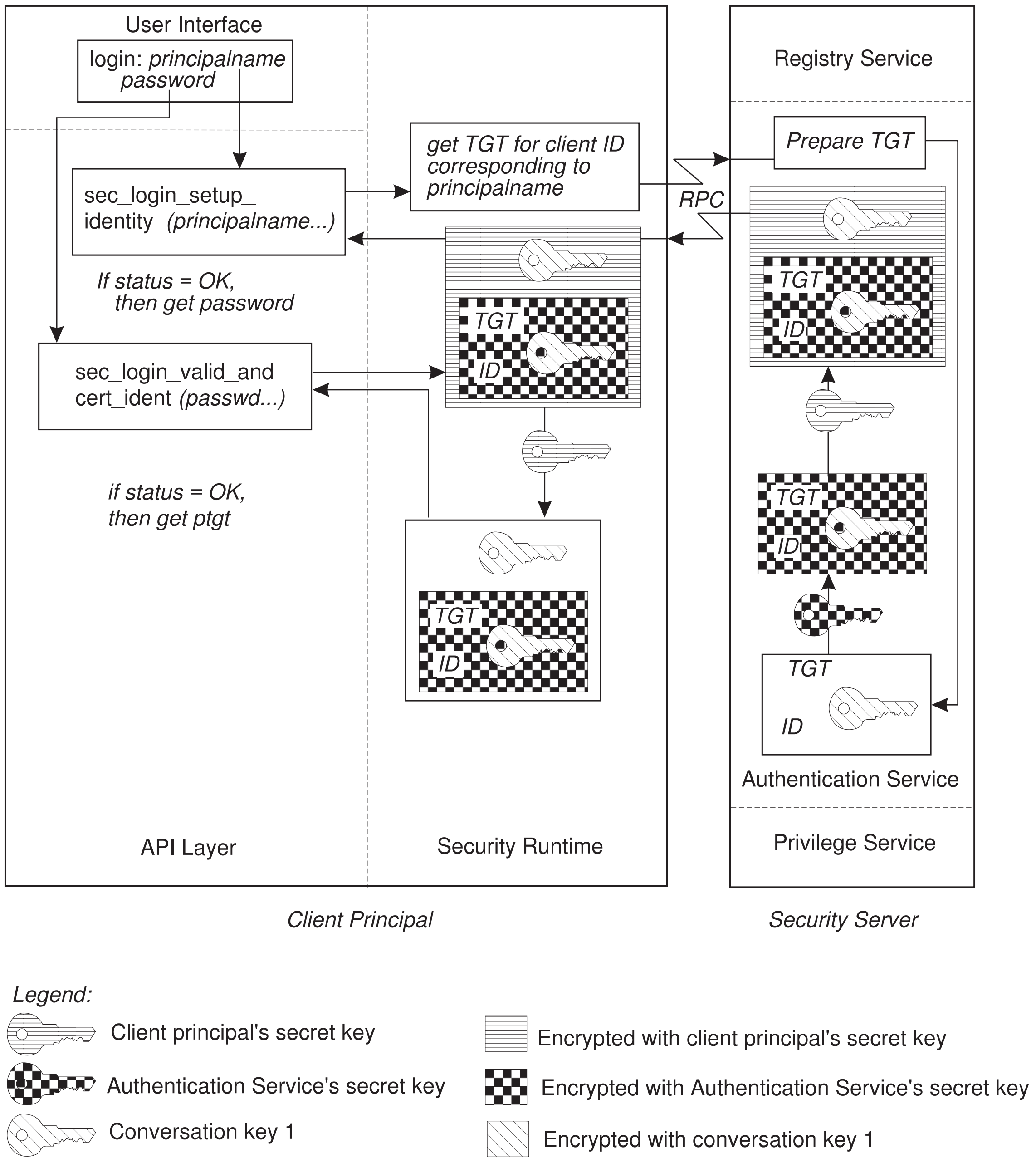

User Interface
login: principalname
password
sec_login_setup_
identity (principalname...)
get TGT for client ID corresponding to principalname
RPC
If status = OK, then get password
sec_login_valid_and
cert_ident (passwd...)
if status = OK, then get ptgt
TGT
ID
Registry Service
Prepare TGT
Authentication Service
Privilege Service
API Layer
Security Runtime
Client Principal
Security Server
Legend:
Client principal's secret key
Authentication Service's secret key
Conversation key 1
Encrypted with client principal's secret key
Encrypted with Authentication Service's secret key
Encrypted with conversation key 1

*Figure 86. Client Acquires Ticket-Granting Ticket Using the OSF DCE 1.0 Protocol*

The OSF DCE 1.0 protocol proceeds as follows (refer to Figure 86 as you read following steps):

1. The user logs in, entering the correct user name. The **dce_login** program invokes **sec_login_setup_identity()**, which takes the user's principal name as one of its arguments. This call causes the client Security runtime to request a Ticket-Granting Ticket (TGT) and passes the user's name (represented as a UUID) to the Authentication Service. A TGT enables a principal to be granted a ticket to a service of interest; in this case, it is the Privilege Service.
2. Upon receiving the request for a TGT, the Authentication Service obtains the user's secret key from the Registry database (where the secret keys of all principals in the cell are stored). Using its own secret key, the Authentication Service encrypts the user's identity, along with a conversation key, in a TGT. The Authentication Service seals the TGT in an “envelope” that is encrypted using the user's secret key. The envelope also contains the same conversation key that is encrypted in the TGT, and is returned to the client.
3. When the TGT envelope arrives, the **dce_login** program prompts the user for the password and invokes **sec_login_valid_and_cert_ident()**. This call passes the password to the local Security runtime library. The Security runtime derives the user's secret key from the password, and uses it to decrypt the envelope. (If the user enters the wrong password, the envelope is undecryptable.) The envelope reveals the conversation key, but the Security runtime cannot decrypt the TGT, since it does not know the Authentication Service's secret key. (A validated TGT is the principal's certificate of identity.)

**Note:** One of the functions of **sec_login_valid_and_cert_ident()** is to demonstrate that the Authentication Service knows the key of the host computer at which the principal is logging in (a server pretending to be the Security server is unlikely to know the host's key). How this is accomplished is not illustrated here, but is explained in Chapter 30, “The Login Application Program Interface” on page 495.

Having acquired the user's Ticket-Granting Ticket, the login program then proceeds with Part 2 of the authentication procedure (described in “How the Client Obtains a Privilege-Ticket-Granting Ticket”).

**How the Client Obtains a Privilege-Ticket-Granting Ticket:** This section describes the acquisition, by the Security Client, of the user's Privilege-Ticket-Granting Ticket. Acquisition of the user's PTGT is the second of the two parts of DCE user authentication.

From this point on, the client principal uses four different conversation keys to talk with other principals. Use of multiple short-lived keys makes an attacker's task far more difficult: there are more encryption keys to discover and less time in which to “crack” them.

**Note:** Refer to Figure 87 on page 423 as you read the following steps.

1. When the Security client runtime has succeeded in decrypting the envelope, the API calls a network layer interface that requests a Privilege-Ticket Granting Ticket (PTGT) from the Privilege Service. For a PTGT to be granted, however, the user must first acquire a ticket to talk to the Privilege Service, which is a principal distinct from the Registry and Authentication Service. The Security runtime therefore requests such a ticket from the Authentication Service. The Security runtime encrypts this request using the conversation key it learned when it decrypted the TGT envelope.
2. Since the request for a ticket to the Privilege Service is encrypted under the conversation key associated with the TGT, the Authentication Service believes that the identity of the user is authentic; that is, no other principal could have sent a message so encrypted because no other principal knows the secret key under which the Authentication Service encrypted that conversation key. Since the user has proved to the Authentication Service knowledge of the key, the Authentication Service allows the user to talk to the Privilege Service, and so prepares a ticket to that service. This ticket contains the identity of the user (and a second conversation key) encrypted under the secret key of the Privilege Service. Like the TGT envelope, the envelope containing the ticket to the Privilege Service also contains the second conversation key, for use in conversing with the Privilege Service, and is encrypted with the first conversation key.

**Note:** Beginning with Figure 87 on page 423 the illustrations do not show the Authentication Service decrypting and reencrypting requests for tickets, since it knows all of the keys.

3. Upon receipt of the envelope containing the ticket to the Privilege Service, the Security client runtime decrypts the envelope using the first conversation key and, in the process, learns the second conversation key. The client RPC runtime sends the Privilege Service ticket to the Privilege Service.

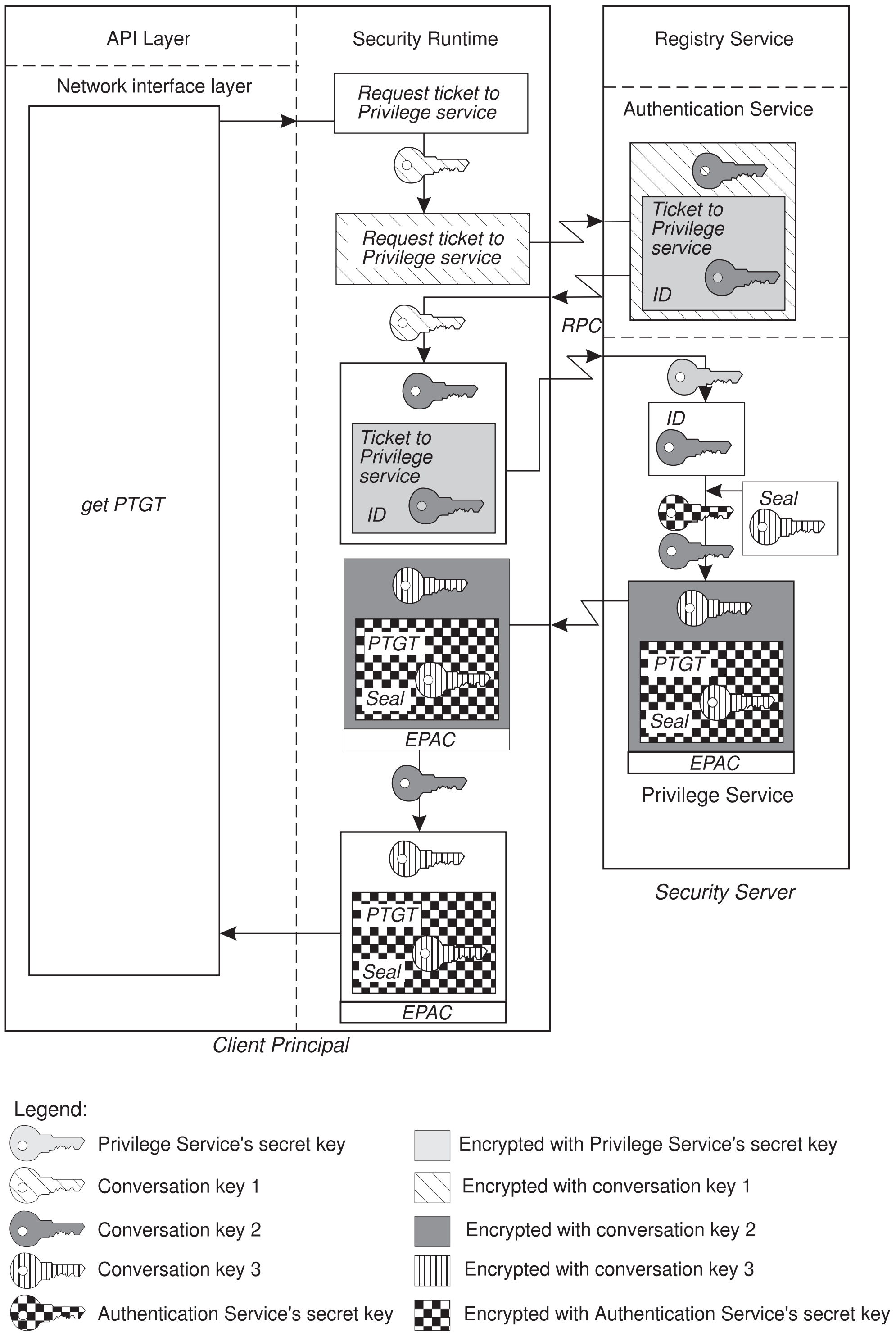

API Layer
Security Runtime
Registry Service
Network interface layer
Request ticket to Privilege service
Authentication Service
Request ticket to Privilege service
Ticket to Privilege service
ID
RPC
Ticket to Privilege service
ID
get PTGT
Seal
PTGT
Seal
EPAC
PTGT
Seal
EPAC
Privilege Service
Security Server
PTGT
Seal
EPAC
Client Principal
Legend:
Privilege Service's secret key
Conversation key 1
Conversation key 2
Conversation key 3
Authentication Service's secret key
Encrypted with Privilege Service's secret key
Encrypted with conversation key 1
Encrypted with conversation key 2
Encrypted with conversation key 3
Encrypted with Authentication Service's secret key

*Figure 87. Client Acquires Privilege-Ticket-Granting Ticket*

4. The Privilege Service decrypts the ticket sent to it, learning both the identity of the user and the conversation key it will use to encrypt its response. The Privilege Service believes the identity is authentic because the ID information was encrypted under its own secret key, and no principal other than the Authentication Service could have encrypted the information using this secret key. Because the Privilege Service trusts the authenticity of the user's identity, it prepares a Extended Privilege Attribute Certificate (EPAC).

   The EPAC describes the user's privilege attributes and any extended attributes that are associated with the user. The EPAC (or EPACs in case of a delegated operation) is sealed with an MD5 checksum. (Delegation is described in Chapter 27, "The Extended Privilege Attribute Application Program Interface" on page 451.) The Privilege Service produces a PTGT that contains the EPAC seal, a third conversation key, and in the case of a delegate operation, the EPAC seal encrypted in the key of the Privilege Server. This encrypted seal is called a *delegation token*. (The Authentication Service and Privilege Service cooperate to prepare the PTGT, although the illustration only shows the Privilege Service preparing it). The EPAC itself is carried outside the PTGT. The EPAC seal is used to verify the integrity of the EPAC data for authenticated RPC calls.

   The PTGT envelope is encrypted using the second conversation key and also includes the third conversation key. (The Authentication Service supplies the third conversation key, although the illustration does not show this detail.)

5. The Security client runtime decrypts the PTGT envelope using the second conversation key, and learns the third conversation key. It cannot decrypt the PTGT itself, since the PTGT is encrypted under the secret key of the Authentication Service.

**The Login Context:** At this point, the Security Server has authenticated the user's identity, and as a result, the user has been able to acquire information about its privilege attributes that the Privilege Service has certified. The client now calls **sec_login_set_context()** to set the login context (a handle to this user's network identity and privilege attributes) to the identity that has been established. Henceforth, processes invoked by this user assume the user's login context, and among these processes is the client side of an application that is the subject of the rest of the walkthrough.

**Identities in a Delegation Chain:** When a user who has initiated delegation (with **sec_login_become_initiator**), makes an authenticated RPC call to the next member in a delegation chain (the intermediary), the initiator passes its EPAC and its PTGT, which contains the seal and delegation token. The intermediary then invokes **sec_login_become_delegate** or **sec_login_become_impersonator**, passes to the Privilege Service the authorization information (EPAC and delegation token) it received from the initiator, and requests the addition of its identity to the delegation chain. The Privilege Service uses the delegation token, which is a seal over the EPAC encrypted in the Privilege Service's key, to determine whether or not to certify the initiator's credentials. If the initiator's credentials are valid, the Privilege Service generates a new seal and delegation token to seal the initiator's and the intermediary's EPACs. Again, the delegation token is encrypted in the Privilege Service's key. The intermediary's authorization information now includes both EPACs in the delegation chain and a PTGT that contains the EPACs seal and delegation token. The subsequent additions of identities to the delegation chain are handled in the same manner, with each intermediary's identity being added to the chain.

## A Walkthrough of DCE Application Authentication

This section explains how DCE Security authenticates an application to which the application developer has added Authenticated RPC calls. It is a continuation of the walkthrough in the previous section and is illustrated in Figure 88 on page 426.

1. Having been authenticated and having acquired a PTGT, the user now runs an application. The client side of the application calls **rpc_binding_import_begin()**, **rpc_binding_import_next()**, and the like. These calls specify the remote interfaces required by the client for the application.
2. The Cell Directory Service returns the client binding handles to the specified interfaces. This example shows the binding model in which the client consults the CDS for the server principal name.
3. The client next sets authorization information for the binding handles by calling **rpc_binding_set_auth_info()**. Among other parameters that it sets, **rpc_binding_set_auth_info()** sets the authentication protocol, the protection level, and authorization protocol for the binding handle corresponding to the remote interface. In this case, assume the following:
   - Authentication protocol (*authn_svc* parameter) is DCE Shared-Secret Authentication.
   - Protection level (*protect_level*) is Packet Privacy. (All RPC argument values are encrypted.)
   - Authorization protocol (*authz_svc*) is DCE Authorization (an EPAC contains UUIDs representing the client's privilege attributes, and the server is most likely to compare this information with the ACLs protecting the objects of interest to determine the principal's authorization).

   The next steps are shown in Figure 89 on page 427.

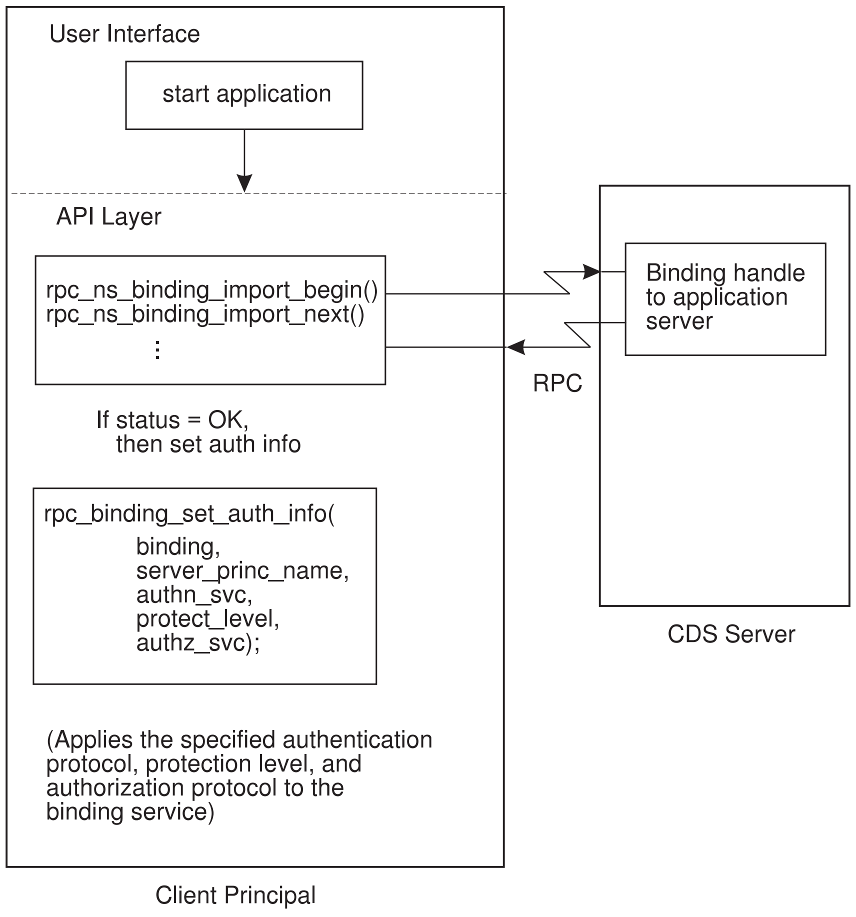

User Interface
start application
API Layer
rpc_ns_binding_import_begin()
rpc_ns_binding_import_next()
⋮
Binding handle to application server
RPC
If status = OK, then set auth info
rpc_binding_set_auth_info( binding, server_princ_name, authn_svc, protect_level, authz_svc);
CDS Server
(Applies the specified authentication protocol, protection level, and authorization protocol to the binding service)
Client Principal

*Figure 88. Client Sets Authentication and Authorization Information*

4. The client now requests an operation to be performed by the server. The client RPC runtime determines the binding handle that corresponds to the remote interface that can perform the operation, and requests a ticket to the principal that supports that interface. To acquire the ticket, the Security runtime encloses the PTGT, along with the principal name of the application server, in an envelope encrypted under the third conversation key. The client sends the envelope to the Authentication Service.
5. The Authentication Service uses the application server's secret key to reencrypt the EPAC and a fourth conversation key. The ticket to the application server is in turn encrypted with the third conversation key in an envelope that also includes the fourth conversation key. The Authentication Service returns the envelope to the client's Security runtime.
6. The Security runtime decrypts the envelope using the third conversation key, in the process learning the fourth conversation key. The Security runtime then uses the fourth conversation key to encrypt the application request to the server, and the client RPC runtime sends the application request to the server.
7. The application server's security runtime receives the client's request, and learns from the header that the request is authenticated.

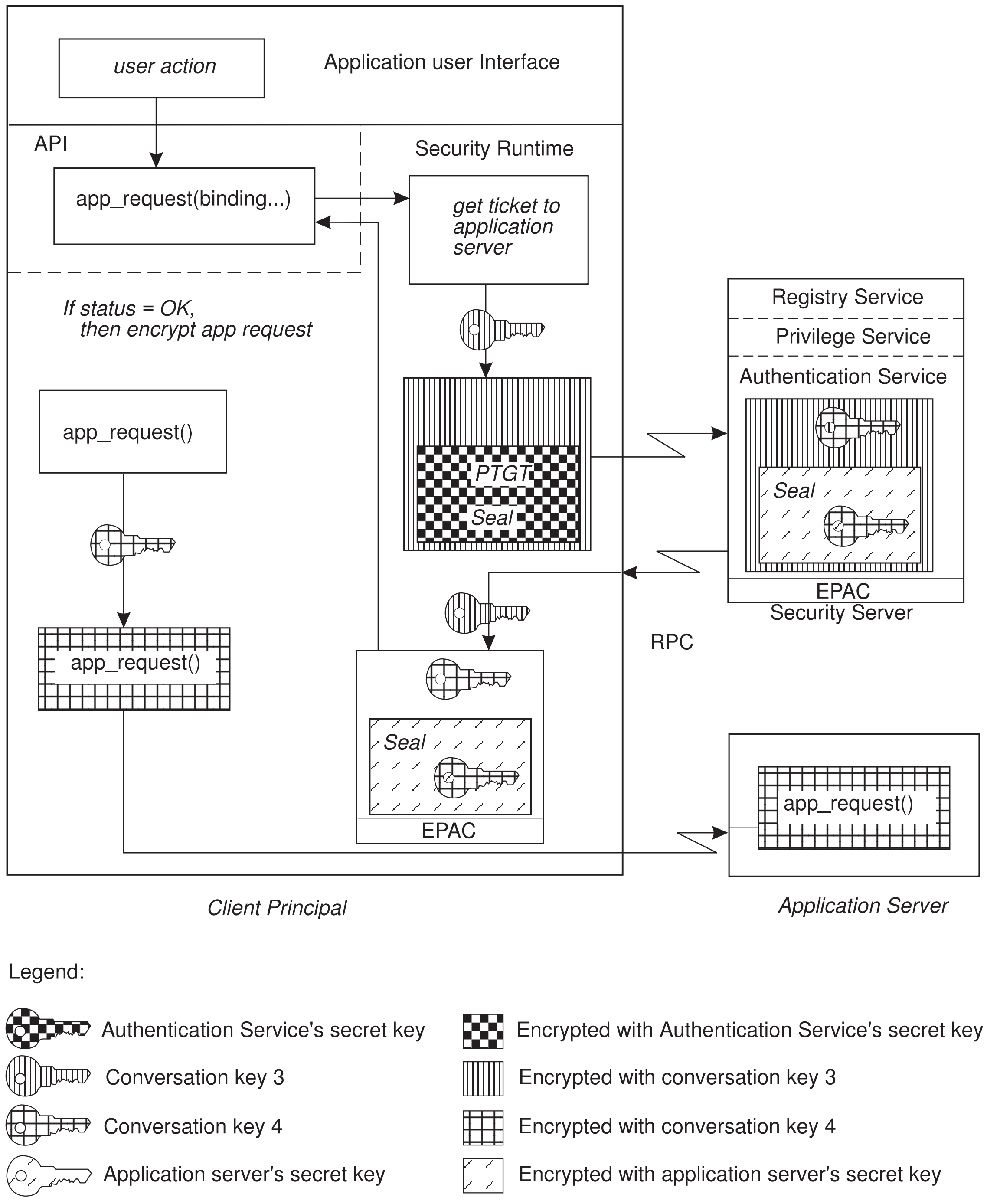

user action
Application user Interface
API
Security Runtime
app_request(binding...)
get ticket to application server
If status = OK, then encrypt app request
app_request()
PTGT
Seal
Registry Service
Privilege Service
Authentication Service
Seal
EPAC
Security Server
RPC
app_request()
Seal
EPAC
app_request()
Client Principal
Application Server
Legend:
Authentication Service's secret key
Conversation key 3
Conversation key 4
Application server's secret key
Encrypted with Authentication Service's secret key
Encrypted with conversation key 3
Encrypted with conversation key 4
Encrypted with application server's secret key

*Figure 89. Client Principal Makes Application Request*

Steps continue in Figure 90 on page 428.

8. Before fulfilling the client's request, there are two things a server needs to do. Namely, it has to learn the conversation key for communicating with the client and authenticate the identity of the client. In order to do this, it sends a *challenge* to the client. To begin the challenge to the client's identity and authorization, the runtime generates a random number and sends it (in plain text) to the client.
9. The Security runtime encrypts the random number using the fourth conversation key, which it received (from the Authentication Service) for the purpose of talking to the application server. The RPC runtime sends the encrypted random number and the server ticket to the application server.

10. The Security runtime decrypts the ticket using its secret key, in the process learning the conversation key and the client's authorization. It uses the conversation key to decrypt the number sent by the client. Because the number is the same random number that the server sent previously, the runtime concludes that the client knows the conversation key, and therefore that the client's identity is authentic.

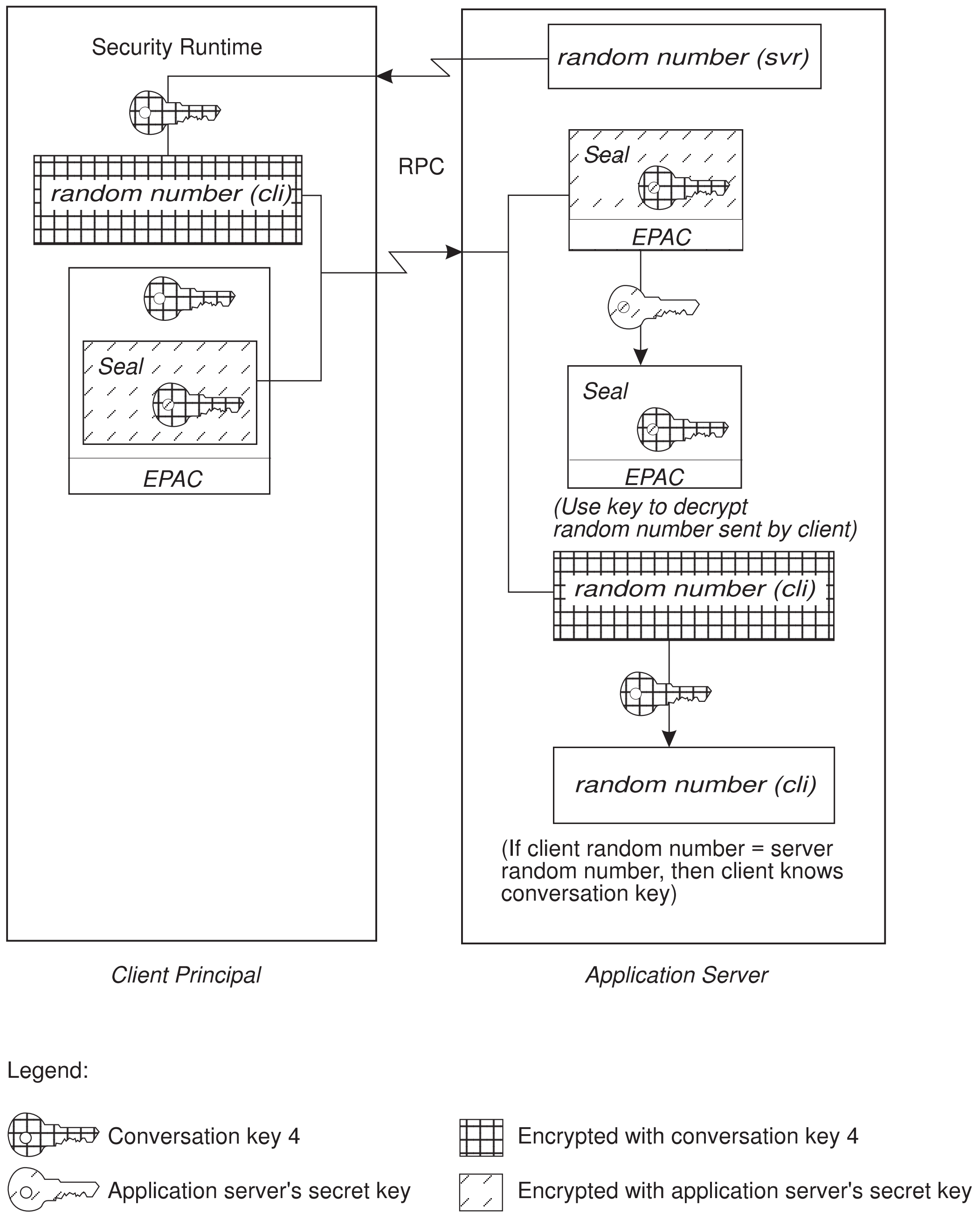

Security Runtime
random number (svr)
RPC
Seal
random number (cli)
EPAC
Seal
EPAC
Seal
EPAC
(Use key to decrypt random number sent by client)
random number (cli)
random number (cli)
(If client random number = server random number, then client knows conversation key)
Client Principal
Application Server
Legend:
Conversation key 4
Application server's secret key
Encrypted with conversation key 4
Encrypted with application server's secret key

Figure 90. Application Server Challenges Client

Steps continue in Figure 91 on page 429.

11. The Security runtime for the application server uses the fourth conversation key to decrypt the client's request. If it determines from the information in the EPAC that the client is authorized, it performs the server operation and prepares a response. The server runtime encrypts the response using the conversation key and sends it to the client.

12. The client runtime receives and decrypts the response, and returns data to the application interface through the API.

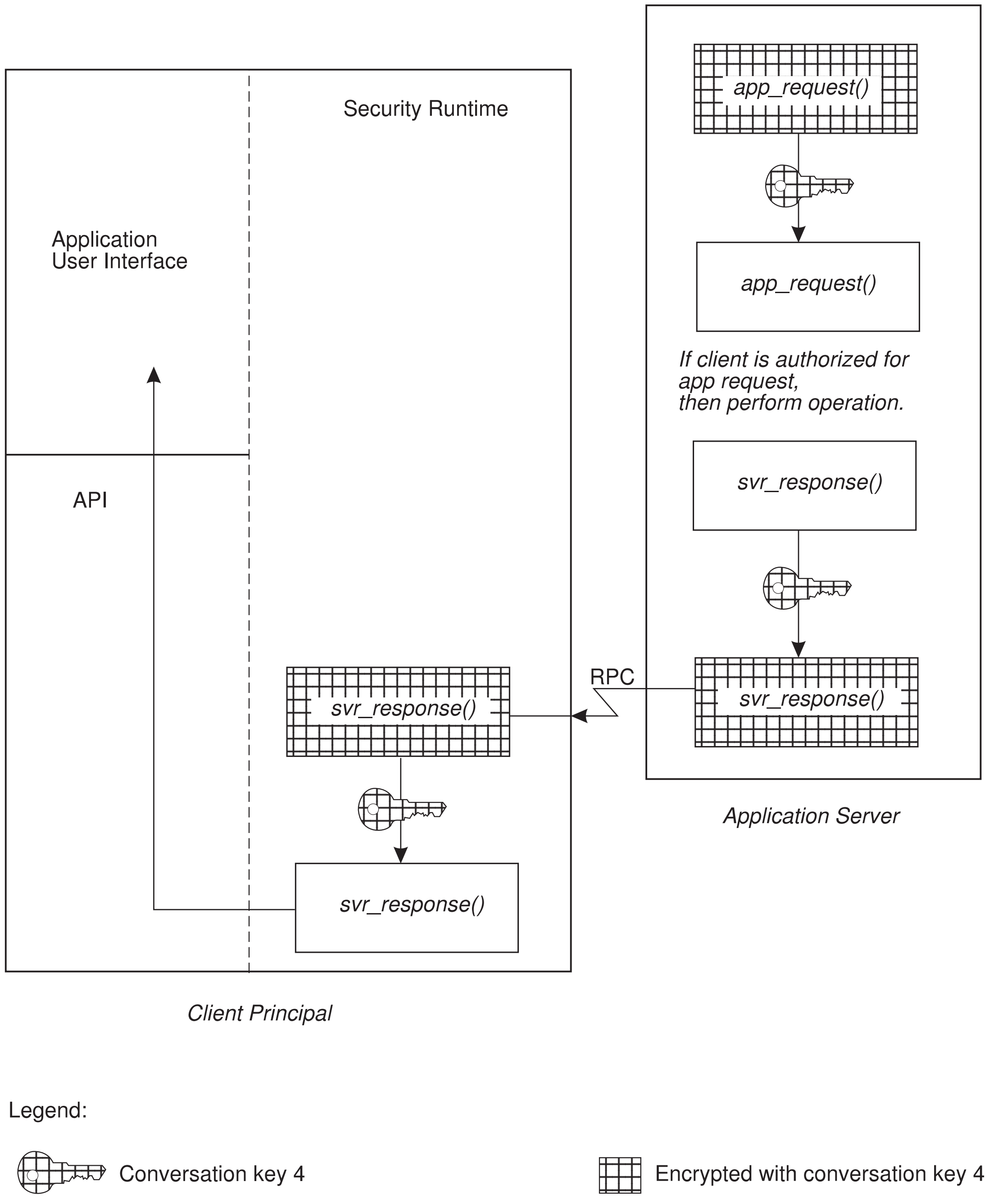


*Figure 91. Application Server Responds to Client's Request*

The application server and client can continue to use the fourth conversation key indefinitely for subsequent conversations. If the server receives an application request after discarding the conversation key, which it may do if it has not heard from client for some time, then the server challenges the client to learn the key, as shown in Figure 90 on page 428. If the client's ticket to the application server expires, it must acquire a new one, as shown in Figure 89 on page 427. If the client wants to talk to a new service, it must acquire a ticket to that service, as shown in Figure 89 on page 427.

**Note:** The illustrations in the walkthrough show the authentication protocol in the context of a datagram-based network communications protocol. In the case of a connection-oriented protocol, the client sends both the application request and the ticket to the server at connection setup, rather than separately (as illustrated in Figure 89 on page 427 and Figure 90 on page 428).

## A Walkthrough of DCE Application Authentication Using GSSAPI

This section describes the process by which applications that manage network communications without RPC can use GSSAPI and DCE Security to authenticate the applications with which they communicate. It continues from the end of “A Walkthrough of User Authentication” on page 416.

In peer-to-peer communications, the application that establishes the secure connection is the *context initiator* or simply *initiator*. The context initiator is like a DCE RPC client. The application that accepts the secure connection is the *context acceptor* or simply *acceptor*. The context acceptor is like a DCE RPC server.

The peer applications establish a secure connection in the following way. The numbers of the stages described correspond to callouts in Figure 92 on page 431.

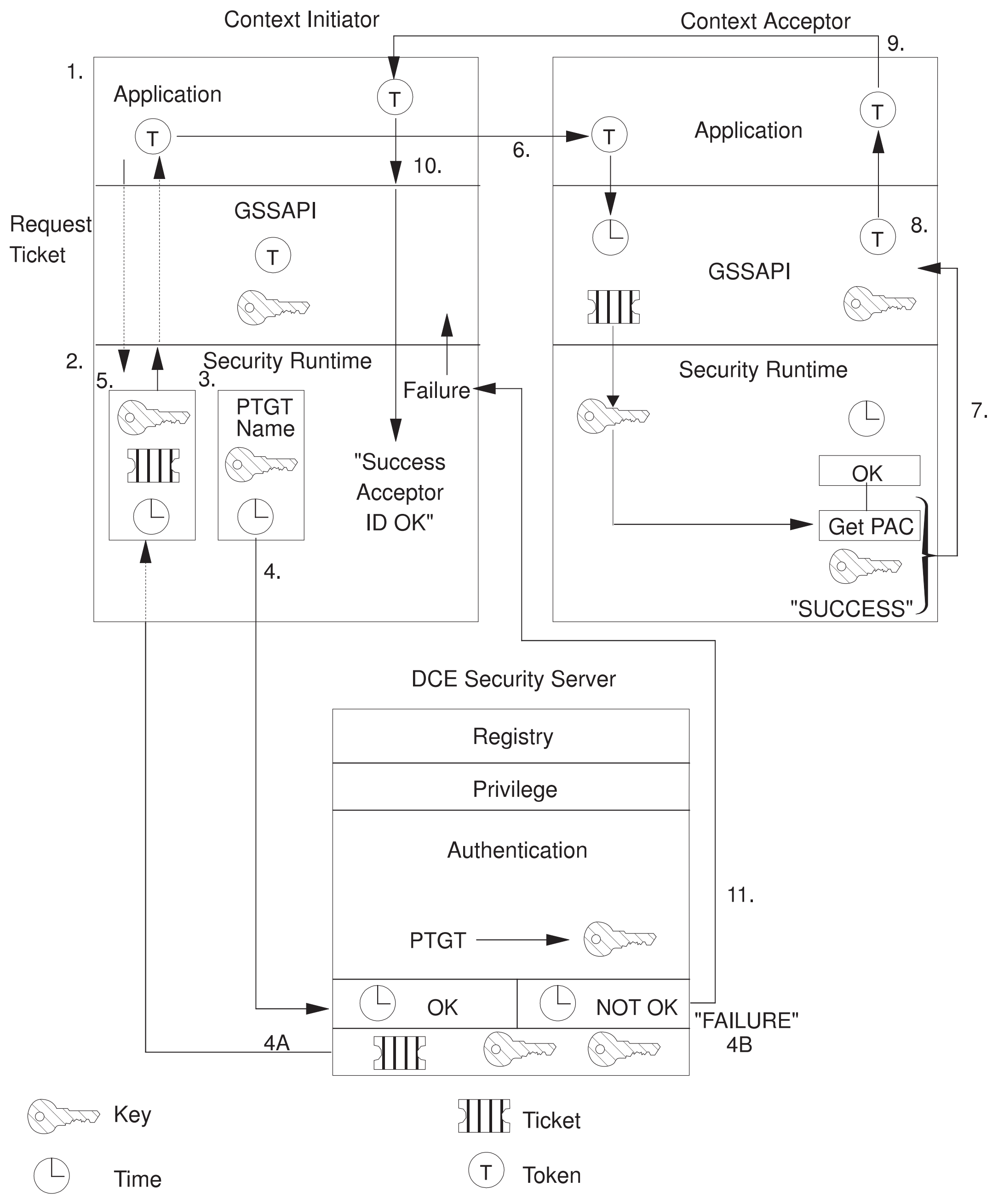


*Figure 92. Peer Applications Establish a Security Context*

1. The context initiator uses the **gss_init_sec_context** routine to request a ticket from the DCE Security Server that will allow the initiator to talk to the context acceptor.

   The initiator's Security Runtime creates an envelope that contains:

   - The initiator's PTGT
   - The acceptor's principal name
   - A time stamp encrypted under the third conversation key

   The Security Runtime sends the envelope to the Authentication Service. The Authentication Service does the following:

   - Decrypts the PTGT to get the third conversation key (see “How the Client Obtains a Privilege-Ticket-Granting Ticket” on page 422).
   - Checks the time encrypted under the third conversation key:
     - If the time does not match, the authentication fails, and the Authentication Service returns a failure message to the Security Runtime, which in turn sends a message to the GSSAPI.
     - If the time matches, the initiator's identity is authenticated, and the Authentication Service creates a ticket to allow the initiator to authenticate itself to the acceptor.

   The ticket consists of the following:

   - A seal of an EPAC(s) which accompanies (that is, not contained within) the ticket.
   - A fourth conversation key, generated by the Authentication Service

   The Authentication Service then encrypts the ticket under the acceptor's secret key. It sends the ticket (encrypted under the acceptor's secret key) and the fourth conversation key (encrypted under the third conversation key) to the initiator's Security Runtime.

2. The initiator's Security runtime:
   - Decrypts the fourth conversation key
   - Encrypts the current time under the fourth conversation key
   - Sends the following to GSSAPI:
     - The ticket
     - The EPAC(s)
     - The encrypted time stamp
     - The fourth conversation key
3. GSSAPI holds onto the fourth conversation key and creates a GSSAPI token containing:
   - Ticket
   - EPAC(s)
   - Encrypted time stamp

   It sends the GSSAPI token to the initiator, which forwards it to the acceptor.
4. The acceptor calls the **gss_accept_sec_context()** routine which separates the ticket, the EPAC(s), and the encrypted time stamp, and send them to the acceptor's Security Runtime.
5. The acceptor's Security runtime:
   - Decrypts the ticket to get the fourth conversation key
   - Checks the time encrypted under the fourth conversation key
   - If the time matches, the Security Runtime:
     - Verifies the seal of the initiator's EPAC(s)
     - Creates a success message and encrypts the message under the fourth conversation key
     - Sends the EPAC(s), the message, and the fourth conversation key to the acceptor's GSSAPI
   - If the time does not match, it sends a failure message to the acceptor's GSSAPI.
6. The acceptor's GSSAPI holds onto the fourth conversation key and the EPAC(s), and creates a GSSAPI token containing the success message. It passes the token to the acceptor.
7. The acceptor forwards the GSSAPI token to the initiator.

8. The initiator passes the token to its GSSAPI to send to the Security Runtime by calling the **gss_init_sec_context()** routine again.
9. The Security Runtime tries to decrypt the message. If it can, it returns a success status to the GSSAPI that the acceptor's identity is authenticated. If not, it returns a failure status to the GSSAPI.

The context acceptor and context initiator can use the fourth conversation key in future communications calling the **gss_sign** and **gss_seal** routines. The context acceptor can get the initiator's EPAC(s) in the form of an **rpc_authz_cred_handle_t** object so it can perform a DCE ACL check by calling the **gssdce_extract_creds_from_sec_context** routine. If the context initiator wants to talk to a new context acceptor, it must acquire a ticket to that context acceptor.

## Intercell Authentication

While the intercell authentication model is an extension of intracell authentication, there are certain concepts that are particular to intercell authentication. This section discusses those concepts.

## Authentication Service Surrogates

A principal trusts another principal in its cell because it trusts the Authentication Service to authenticate all principals that are members of the cell, except for the Authentication Service itself, which its member principals trust *a priori*. The Authentication Service can authenticate all principals in its cell because it shares a secret key with each of them. A principal that wants to talk to a foreign principal (that is, a principal in another cell) must acquire a ticket to that principal. Furthermore, the ticket must be encrypted in the secret key of the foreign principal, or else the foreign principal may disregard the initiator of the conversation. The local principal cannot get such a ticket from its own Authentication Service because the local Authentication Service does not know the secret keys of any foreign principals. Therefore, there must be some other means by which the two instances of the Authentication Service can securely convey information about their respective principals to one another.

**The Problem:** Besides the fact that it is trusted *a priori*, a cell's Authentication Service principal is an exception in another respect. Other kinds of principals share their secret keys with the local Authentication Service, whereas the Authentication Service's key is private (that is, not known to another principal). Thus one problem of intercell authentication to be overcome is how the Authentication Service in one cell communicates securely with that in another, without either of them having to share their private keys. Sharing the keys would introduce an unacceptable security risk.

**Note:** The Kerberos network authentication specification makes a distinction between the terms *secret* and *private*. Secret refers to data that is known to two principals, and private refers to data that is known to only one principal. The same distinction is made in this book.

**The Solution:** The solution to this problem is an extension of the Shared-Secret Authentication model previously discussed in this chapter. An entry in the Registry database of one cell specifies the same secret key as that in an entry in the other cell's Registry database. The two Registry database entries are known as mutual authentication surrogates, and the two cells that maintain mutual authentication surrogates are called trust peers. It is through their surrogates that two instances of the Authentication Service convey information about their respective principals to one another, enabling a principal from one cell to acquire a ticket to a principal in another cell.

An authentication surrogate is a principal in the sense that it is represented by an entry in a Registry database, but it is not an autonomous participant in authenticated communications in the same sense as a user or a server. Rather, it is like an alias that is assumed by a cell's Authentication Service when it communicates with a trust peer. A trust peer relationship established between two cells is an implicit expression of mutual trust between the two Authentication Services on the part of the cell administrators who establish the relationship. (Administrators use the **rgyedit** program to establish the relationship.)

## Intercell Authentication by Trust Peers

This section explains how a client principal in one cell is authenticated by an Authentication Service in a peer cell so the client principal may communicate with another principal that is a member of the foreign cell.

1. A client principal, having already been authenticated by its Authentication Service and acquired its EPAC, requests a service from a foreign cell. The client specifies the server principal that provides the service by its fully qualified name (which identifies the foreign cell as well as the cell-relative server principal name).
2. Recognizing by its name that the server principal is foreign, the client's Security runtime makes a request to the local Authentication service for a TGT to the Authentication service of the foreign cell of which the server principal is a member. The request for the foreign TGT (FTGT) proceeds like a ticket-granting request for any other target principal. The local Authentication Service constructs the ticket, preserving EPAC data from client's existing PTGT, and encrypts it using the secret key that the two Authentication surrogates share.
3. On receiving the request for the FTGT, the foreign Authentication Service decrypts it using the surrogates' secret key, and returns a ticket to the foreign Privilege Service to the client's Security runtime.
4. The client's Security runtime uses the ticket to the foreign Privilege Service to obtain a Foreign Privilege-Ticket-Granting Ticket (FPTGT). The FPTGT is simply the client's original EPAC encrypted with the key of the foreign Privilege Service.
5. After the client principal receives the FPTGT, it requests a ticket to the foreign server principal from the foreign Authentication Service, exactly as it would request a ticket to a local principal from its own Authentication Service. The client principal may also reuse the FPTGT to the foreign cell to acquire tickets to any other principals in that cell.

# Chapter 25. Authorization

This chapter explains concepts related to authorization. With the Authenticated RPC facility, you select the authorization protocol your application uses. Among the authorization protocols supported by z/OS DCE Security for use by Authenticated RPC are DCE Authorization (the default), and Name-Based Authorization.

This chapter first discusses DCE Authorization, and more particularly, DCE Access Control Lists (ACLs). Later it discusses the Name-Based Authorization protocol.

## DCE Authorization

The DCE Authorization protocol is based in part on the UNIX system file-protection model, but is extended with ACLs. An ACL is a list of access control entries that protects an object. Each entry in the ACL specifies a set of *permissions*. Usually, most of the entries in the ACL specify a privilege attribute (such as membership in a group) and the set of permissions that can be granted to the principals possessing that privilege attribute. Other entries specify a set of permissions that can ***mask*** the permission set in a privilege attribute entry.

Every ACL is managed by an ACL manager type. It determines a principal's authorization to perform an operation on an object by reading the object's ACL to find the appropriate entry (or entries) that matches some privilege attribute possessed by the principal. If the type of access requested by the principal is one of the permissions listed in the matching entry (and assuming no applicable mask entry denies that permission), then the ACL manager type allows the principal to perform the requested operation. If the requested permission is not listed in the matching ACL entry, or is denied by a mask, permission to perform the operation is denied. Permission is also denied if the ACL contains no matching privilege attribute entry.

Unlike UNIX system file permissions, DCE ACLs are not limited to the protection of file system objects (that is, files, directories, and devices). ACLs can also control access to nonfile-system objects, such as the individual entries in a database.

**Note:** The implementation of DCE ACLs is aligned with POSIX P1003.6 Draft 12.

In this chapter, the general term *name* refers to a principal, group, or cell identifier. These names have two representations:

- As UUIDs in ACL program interfaces
- As print strings in user interfaces.

### Object Types and ACL Types

The ACL facility distinguishes between two types of objects: *container objects* and *simple objects*. Container objects contain other objects, which can be simple or other container objects. Simple objects do not contain other objects. Examples of container objects include a filesystem directory or a database; examples of simple objects include a file or a database entry.

To protect both object types and to enable newly created objects to inherit default ACLs from their parent container objects, the ACL facility supports two basic kinds of ACLs:

- An Object ACL is associated with either a container or a simple object, and controls access to it.

- A Creation ACL is associated with a container object only. Its function is not to control access to the container, but to supply default values for the ACLs of objects created in the container. There are two types of Creation ACLs:
  - An Initial Object Creation ACL supplies default values for a simple object's Object ACL and for a container object's Initial Object Creation ACL.
  - An Initial Container Creation ACL supplies default values for both a container object's Object ACL and its Initial Container Creation ACL.

Figure 93 illustrates how ACL defaults are derived from Creation ACLs.

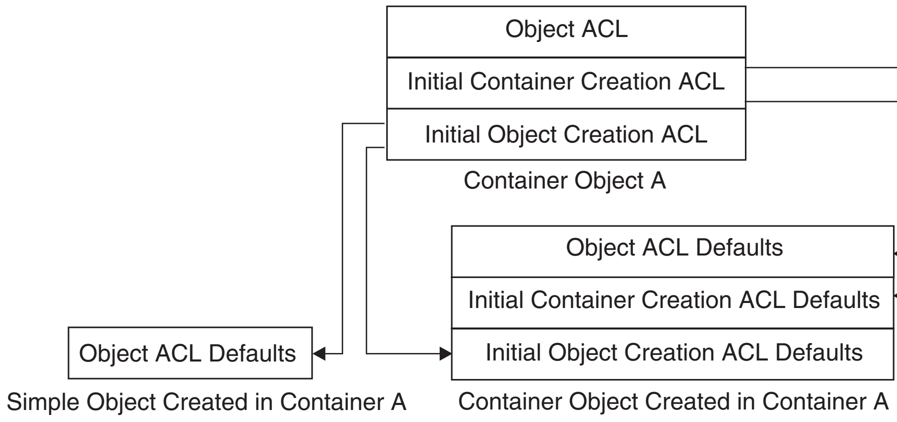

Object ACL
Initial Container Creation ACL
Initial Object Creation ACL
Container Object A
Object ACL Defaults
Initial Container Creation ACL Defaults
Initial Object Creation ACL Defaults
Object ACL Defaults
Simple Object Created in Container A
Container Object Created in Container A

*Figure 93. Derivation of ACL Defaults*

Aside from the distinctions described previously, there are no differences between Object ACLs and Creation ACLs. Therefore, the information about ACLs in the rest of this chapter does not differentiate between them.

## ACL Manager Types

A separate ACL manager type manages the ACLs for each class of objects for which permissions are uniquely defined. In this context, *class* refers to a category into which objects are placed on the basis of both their purpose and their internal structure. The manager type defines the permissions for those objects whose ACLs it manages:

- The number of permissions
- The meanings of the permissions
- The tokens that represent the permissions in user interfaces to ACL manipulation tools (the default DCE tool is **dccp**).

For example, for access control, five classes of objects are defined in the Registry database, and five ACL manager types manage the ACLs for Registry database objects. (The five Registry manager types run in a single Security Server process.) Other DCE components have their own manager types, and applications layered on DCE can also use manager types for the objects the applications protect.

Refer to z/OS DCE administration documentation for information about standard DCE ACL manager types and the permissions they put into effect. In this book, refer to Chapter 32, "The Access Control List Application Program Interfaces" on page 505 for information about using ACL manager types for distributed applications.

## ACLs

An ACL consists of:

- An ACL manager type identifier, which identifies the manager type of the ACL.
- A default cell identifier, which specifies the cell where a principal or group identified as local is assumed to be a member. A DCE global path name is necessary to specify a principal or a group from a nondefault cell; this consists of a pair of UUIDs representing the principal or group, and the cell of which it is a member. It is necessary to use the ID Map API to convert the global print string names of foreign principals and groups to the UUID representations that DCE ACL managers recognize. (Refer to Chapter 33, "The ID Map Application Program Interface" on page 515 for more information on this subject.)
- At least one ACL entry.

The rest of this chapter discusses ACLs primarily from a user-interface point of view. This perspective provides an orientation to the discussion of the ACL API in this section.

## ACL Entries

DCE Authorization defines the following two basic kinds of ACL entries:

1. Those that associate a specified privilege attribute with a permission set: these are privilege attribute entries.
2. Those that specify a permission set that masks a permission set specified in a privilege attribute entry: these are mask entries.

The following sections describe the two kinds of ACL entries in detail.

**Privilege Attribute Entry Types:** Following are descriptions of the ACL entry types that specify privilege attributes. The privilege attributes of a principal are based on identity and include the principal's name, its group memberships, and local cell. Not all ACL manager types use all privilege attribute entry types. For example, the ACL manager type of a database object probably would not support the **user_obj** and **group_obj** entry types.

**Note:** The term *local cell* means the cell specified in the ACL entry, this is not necessarily the cell in which the protected object resides.

The following are descriptions of the ACL entry types (an ACL entry type is a field in the ACL entry where you can define entries for principals, groups and masks) that specify privilege attributes:

| | |
|---|---|
| **user_obj** | Establishes the permissions for the object's *user*; that is, the user that created the object. An ACL can contain only one entry of this type. The identity of the principal to which this ACL entry refers is assumed to be local and is specified somewhere other than in this entry. In the case of a file, for example, the identity is attached to the file's inode. |
| **user** | Establishes the permissions for the local principal named in this entry. An ACL can contain a number of entries of this type, but each entry must be unique with respect to the principal it specifies. |
| **foreign_user** | Establishes the permissions for the foreign principal named in this entry. An ACL can contain a number of entries of this type, but each entry must be unique with respect to the foreign principal it specifies. This entry type is exactly like the **user** entry type except that this entry explicitly names a cell (for the entry type **user**, the principal inherits the cell specified by the default cell identifier in the ACL header). |

| | |
|---|---|
| **group_obj** | Establishes the permissions for the object's *group*, that is, the group that created the object. An ACL can contain only one entry of this type. As with the **user_obj** entry, the identity of the group is assumed to be local and is specified elsewhere than in the **group_obj** entry itself. |
| **group** | Establishes the permissions for the local group named in this entry. An ACL can contain a number of entries of this type, but each entry must be unique with respect to the group it specifies. |
| **foreign_group** | Establishes the permissions for the foreign group named in this entry. An ACL can contain a number of entries of this type, but each entry must be unique with respect to the foreign group it specifies. This entry type is exactly like the **group** entry type except that this entry explicitly names a cell. (For the entry type **group**, the principals inherit the default cell identifier.) |
| **other_obj** | Establishes the permissions for local principals whose identities do not correspond to any entry type that explicitly names a principal or group; an ACL can contain only one entry of this type. |
| **foreign_other** | Establishes the permissions for all principals that are members of a specified foreign cell and whose identities do not correspond to any **foreign_user** or **foreign_group** entry. An ACL can contain a number of entries of this type, but each entry must specify a different foreign cell. |
| **any_other** | Establishes the permissions for principals whose privilege attributes do not match those specified in any other entry type. An ACL can contain only one entry of this type. |

The following additional ACL entry types are supplied for delegated identities:

- **user_obj_delegate**
- **user_delegate**
- **foreign_user_delegate**
- **group_obj_delegate**
- **group_delegate**
- **foreign_group_delegate**
- **foreign_other_delegate**
- **other_obj_delegate**
- **foreign_other_delegate**
- **any_other_delegate**

These ACL entry types are described in detail in Chapter 27, "The Extended Privilege Attribute Application Program Interface" on page 451 along with the extensions to the ACL checking algorithm for delegation.

ACL entries for privilege attributes consist of three fields in the following form:

*entry_type[:key]:permissions*

Following are descriptions of the fields:

- The ACL *entry type* specifies an ACL entry type as described in "Privilege Attribute Entry Types" on page 437.
- The *key* field specifies the privilege attribute to which the permissions listed in the entry apply. The key field for the ACL entry types **user**, **group**, **foreign_user**, **foreign_group**, and **foreign_other**

explicitly names a principal, group,or cell. For the entry types **foreign_user**, **foreign_group**, and **foreign_other**, the key field must contain a global DCE pathname of the following forms:

– /.../*cellname/principalname*,

– /.../*cellname/groupname*,

– /.../*cellname*,

respectively. The entry types **user_obj, group_obj, other_obj,** and **any_other** do not use the key field.

- The *permissions* field lists the permissions that can be granted to the principal possessing the privilege attribute specified in the entry, unless a mask (or masks) further restricts the permissions that can be granted to the principal. The number and meaning of the permissions that can protect an object are defined by the object's ACL manager type. Therefore, the permissions that an ACL entry can specify must be the set, or a subset, of the permissions defined by the manager type of the ACL in which the entry appears. An ACL manager type implements a specific set of permissions that a principal can perform; a different ACL manager type implements a different permission set.

  A principal is denied access when a **user** or **foreign_user** entry that names the principal contains an empty permission set.

## Mask Entry Types:

| | |
|---|---|
| **mask_obj** | Establishes the permission set that masks all privilege attribute entry types except the **user_obj** and **other_obj** types. |
| **unauthenticated** | Establishes the permission set that masks the permission set in a privilege attribute entry that corresponds to a principal whose privilege attributes have not been certified by an authority such as the Privilege Service. |

The two masks are similar in this respect: the permission set specified in the mask entry is intersected with the permission set in a privilege attribute entry (using a logical AND). This masking operation produces the ***effective permission*** set — that is, the permissions that can be granted to the principal—for the principal possessing the privilege attribute. For example, if a privilege attribute entry specifies the permissions **ab** and a mask entry that specifies the permissions **bc** masks that privilege attribute entry, the effective permission set is **b**. Similarly, a mask entry that specifies the empty permission set means that none of the permissions in any privilege attribute entry that mask entry masks is granted to the principal possessing the privilege attribute.

The two masks are dissimilar in one notable respect. Adding an **unauthenticated** mask entry with an empty permission set to an ACL is equivalent to omitting the **unauthenticated** mask entry from the ACL. In both cases, the set of effective permissions for principals possessing unauthenticated privilege attributes is empty. However, adding a **mask_obj** entry with an empty permission set to an ACL is different from having no **mask_obj** entry in the ACL. In the first case, the effective permission set is empty; in the second, the effective permission set is identical to the permission set in the privilege attribute entry.

ACL entries for masks consist of two fields in the following form:

*entry_type:permissions*

- The *entry_type* field specifies one of the two masks entry types: **mask_obj** or **unauthenticated.**
- The *permissions* field specifies the permission set that masks the permission set in any privilege attribute entry masked by the mask entry.

**The Extended ACL Entry Type:** The ACL entry type **extended** is a special entry type for ensuring the compatibility of ACL data created by different software revisions. It enables old application clients to copy ACLs from one newer revision object store to another without losing data. It also enables obsolete clients to manipulate ACL data that they understand without corrupting the extended entries that they do not understand.

## Access Checking

Standard DCE ACL manager types use a common access-check algorithm to determine the permissions they grant to a principal. Access checking is performed in up to six stages, in the following order:

1. The **user_obj** entry check
2. The check for a matching **user** or **foreign_user** entry
3. The **group_obj** entry check and the check for matching **group** or **foreign_group** entries
4. The **other_obj** entry check
5. The check for a matching **foreign_other** entry
6. The **any_other** check

If, during any stage of access checking, an ACL manager type finds a privilege attribute entry matching a privilege attribute possessed by a principal, the manager type does not proceed with any subsequent stages. This occurs even though the principal can possess other privilege attributes for which there are other matching entries. The following subsections describe the algorithms used at each stage of access checking.

**The user_obj Entry Check:** The pseudocode in Figure 94 illustrates the **user_obj** check algorithm. If the principal seeking access is the identity to which the **user_obj** entry refers, the remaining checks are not performed.

```
IF (no USER_OBJ principal name is available)
THEN
  the requested permission is denied
ELSE IF (the principal name matches the user name associated
      with the USER_OBJ entry) AND (the cell name matches
      the cell name for that entry)
THEN
  IF (the requested permission is listed in the USER_OBJ entry)
  THEN
    IF (the principal's privilege attributes are certified)
    THEN
      the requested permission is granted
    ELSE
      IF (the requested permission is listed in the
       unauthenticated mask entry)
      THEN
        the permission is granted
      ELSE
        the permission is denied
      ENDIF
    ENDIF
  ENDIF
ELSE
  the permission is denied
ENDIF
```

*Figure 94. Pseudocode to Illustrate the user_obj Entry Check*

**The User Entries Check:** The pseudocode in Figure 95 illustrates the algorithm for checking **user** or **foreign_user** entries. If the principal's identity matches one of these entries, the remaining checks are not performed.

```
IF (the principal name matches the user name of any USER
      or FOREIGN_USER entry) AND (the principal's cell name
      matches the cell name for that entry)
THEN
  IF (the requested permission is listed in the USER or
     FOREIGN_USER entry) AND ((the requested permission
     is listed in the mask_obj entry) OR (there is no
     mask_obj entry))
  THEN
    IF (the principal's privilege attributes are certified)
    THEN
      the requested permission is granted
    ELSE
      IF (the requested permission is listed in the
         unauthenticated mask entry)
      THEN
        the permission is granted
      ELSE
        the permission is denied
      ENDIF
    ENDIF
  ELSE
    the permission is denied
  ENDIF
ENDIF
```

*Figure 95. Pseudocode for the foreign_user check*

**The Group Entries Check:** The pseudocode in Figure 96 on page 442 illustrates the algorithm for checking group entries. If a principal is associated with a concurrent group set, more than one search of the ACL entries for groups is performed: one for the primary group (the one specified in the principal's account information) and one for each group in the concurrent group set.

The permissions granted are the union (the logical OR operation) of the permissions produced by each search of the group entries. For example, if two groups where an authenticated principal is a member specify the permission sets **abc** and **cde**, the principal is granted the permission set **abcde**.

If one or more matching group entries are found, the remaining checks are not performed.

```
IF (a group name among the principal’s privilege
      attributes matches the group ID of any GROUP_OBJ, GROUP,
      or FOREIGN_GROUP entry) AND (the principal’s cell name
      matches the cell name for that entry)
THEN
  IF (the requested permission is listed in the group entry)
     AND ((the requested permission is listed in the
     mask_obj entry) OR (there is no mask_obj entry))
  THEN
    IF (the principal’s privilege attributes are certified)
    THEN
      the permission is granted
    ELSE
      IF (the requested permission is listed in the
         unauthenticated mask entry)
      THEN
        the permission is granted
      ELSE
        the permission is denied
      ENDIF
    ENDIF
  ELSE
    the permission is denied
  ENDIF
ENDIF
```

*Figure 96. Pseudocode for Checking Group Entries*

**The other_obj Entry Check:** The pseudocode in Figure 97 illustrates the algorithm for checking the **other_obj** entry.

```
IF (the requested permission is listed in the OTHER_OBJ entry
      AND (the principal’s cell name matches the cell name for
      that entry)
THEN
  IF (the principal’s privilege attributes are certified)
  THEN
    the permission is granted
  ELSE
    IF (the requested permission is listed in the
       unauthenticated mask entry)
    THEN
      the permission is granted
    ELSE
      the permission is denied
    ENDIF
  ENDIF
ELSE
  the permission is denied
ENDIF
```

*Figure 97. Pseudocode to Check the other_object Entry*

**The foreign_other Entries Check:** The pseudocode in Figure 98 illustrates the algorithm for checking the **foreign_other** entries.

```
IF (the requested permission is listed in a FOREIGN_OTHER
      entry) AND (the principal's cell name matches the cell name
      for that entry) AND ((the requested permission is listed
      in the mask_obj entry) OR (there is no mask_obj entry))
THEN
  IF (the principal's privileges are certified)
  THEN
    the permission is granted
  ELSE
    IF (the requested permission is listed in the
       unauthenticated mask entry)
    THEN
      the permission is granted
    ELSE
      the permission is denied
    ENDIF
  ENDIF
ELSE
  the permission is denied
ENDIF
```

*Figure 98. Pseudocode to Check the foreign_other Entries*

**The any_other Entry Check:** The pseudocode in Figure 99 illustrates the **any_other** check algorithm. If no privilege attribute possessed by a principal matches any entry checked in any preceding stage of access checking, the principal can be granted the effective permissions produced by this check.

**Note:** If an ACL listing this entry also lists the **other_obj** entry, only undistinguished foreign identities can match this entry. However, if the ACL does not list the **other_obj** entry, all undistinguished identities, whether foreign or local, match this entry.

```
IF (the requested permission is listed in the any_other entry
      AND ((the requested permission is listed in the mask_obj
      entry) OR (there is no mask_obj entry))
THEN
  IF (the principal's privilege attributes are certified)
  THEN
    the permission is granted
  ELSE
    IF (the requested permission is listed in the
       unauthenticated mask entry)
    THEN
      the permission is granted
    ELSE
      the permission is denied
    ENDIF
  ENDIF
ELSE
  the permission is denied
ENDIF
```

*Figure 99. Pseudocode to Check the any_other Entry*

## Examples of ACL Checking

This section discusses some examples that illustrate ACLs and the access-check algorithms. The examples use the arbitrary convention of ordering entries as follows: masks, principals, groups, *other* entries. However, the access check algorithm disregards the order in which entries appear in an ACL. The permissions in these examples do not refer to any particular permissions defined by any ACL manager type.

**Example 1:** Following is an ACL that protects an object to which three principals, **janea**, **/.../cella/fritzb**, and **mariac**, seek access:

```
mask_obj:ab
user_obj:abc
user:janea:abdef
foreign_user:/.../cella/fritzb:abc
group:projectx:abcf
group:projecty:bcg
```

**Note:** The numbered lists in the discussions that follow correspond to stages 1, 2, 3, 4, 5 and 6 of the access-check algorithm described in "Access Checking" on page 440.

The principal **janea** requests permission **c** to the object protected by the ACL. Assume that the principal **janea** has the privilege attributes of being a member of the groups **projectx** and **projecty** (as well as having a **user** entry that names her) and that **janea** is the principal to which the **user_obj** entry refers. Assume also that this principal's privilege attributes are certified:

- The **user_obj** check produces the permissions **abc**.

The result of this check is that the effective permission set for **janea** is **abc**. Because a matching entry is found during the first stage of access checking, none of the remaining stages of access checking is performed even though there are three other matching entries. The **mask_obj** entry does not mask the **user_obj** entry, so **janea**'s effective permissions are the permissions in the **user_obj** entry. Because **janea** requested a permission that is a member of the effective permission set, her request is granted.

The second principal seeking access to the protected object is **/.../cella/fritzb**. This principal requests permission **b**. Assume that **user_obj** resolves to some identity other than **/.../cella/fritzb,** and that this principal's privilege attributes are uncertified:

1. The **user_obj** check produces no permissions because **/.../cella/fritzb**'s identity does not match that of the **user_obj** (no foreign principal can ever match this entry).
2. The **foreign_user** entry for **/.../cella/fritzb** specifies the permissions **abc**. The application of the **mask_obj**, which specifies the permissions **ab** to this permission set, produces the permissions **ab**. Because the **unauthenticated** mask entry is missing from the ACL, all permissions for unauthenticated identities are masked, producing an empty effective permission set.

The result of these checks is that **/.../cella/fritzb**'s request is denied (and would be denied, regardless of the permission requested). Only the first two stages of access checking are performed.

The third principal seeking access is **mariac**, who requests permission **a**. Assume that the privilege attributes of **mariac** are certified, that **mariac** is not the principal that corresponds to the **user_obj** entry, and that **mariac** is a member of the groups **projectx** and **projecty**:

1. The **user_obj** check produces no permissions.
2. There is no matching user entry.
3. The group check finds two matching entries. The permissions associated with **projectx** (**abcf**) when masked by the **mask_obj** entry (**ab**) produce the permissions **ab**. The permissions associated with

**projecty** (**bcg**) when masked by the **mask_obj** entry produce the permission **b**. The union of the permission sets **ab** and **b** is the set **ab**.

The effective permission set for **mariac** is **ab**, and because the requested permission (**a**) is a member of that set, **mariac**'s request is granted. The remaining stages of access checking are not performed.

**Example 2:** Following is the ACL for an object to which two principals, **ugob** and **/.../cellb/lolad**, seek access:

```
mask_obj:bcde
unauthenticated:ab
user_obj:abcdef
user:ugob:abcdefg
group:projectz:abh
foreign_other:/.../cellb/:abc
```

**Note:** The numbered lists in the discussions that follow correspond to stages 1, 2, 3, 4, 5 and 6 of the access check algorithm described in "Access Checking" on page 440.

The principal **ugob** requests permission **b.** Assume that **ugob** is not the principal the **user_obj** entry refers to. Assume also that the privilege attributes of **ugob** include membership in the group **projectz**, in addition to the **user** entry that names him. The principal has failed to acquire certified privilege attributes:

1. The **user_obj** check produces no permissions.
2. The matching entry among the user entries specifies the permissions **abcdefg.** Applying **mask_obj** (**bcde**) produces the permission set **bcde**. Applying the **unauthenticated** mask (**ab**) to the permission set **bcde** produces the effective permission set **b**.

Because the principal **ugob** requests a permission (**b**) that is a member of the effective permissions set, this principal's request is granted.

A case that illustrates how access is determined for otherwise undifferentiated members of a specified foreign cell is that of the principal **/.../cellb/lolad**, who requests permission **e**. Assume that the privilege attributes of this principal are certified:

1. The principal is foreign, so the **user_obj** check cannot be a match.
2. There are no **foreign_user** entries.
3. There are no **foreign_group** entries.
4. **lolad** is a member of **cellb**, meaning that the privilege attributes match those in the **foreign_other** entry for **cellb.** The permissions specified by the **foreign_other** entry for **cellb** (**abc**) as masked by **mask_obj** (**bcde**) produces the effective permission set **bc**.

The permission requested (**e**) is not a member of the effective permission set (**bc**), so the request is denied.

**Example 3:** Following is the ACL for an object to which one principal, **silviob** seeks access.

```
unauthenticated:a
user:jeand:abcde
user:denisf:-
group:projectx:abcd
foreign_other:/.../cella:-
foreign_other:/.../cellc:abc
any_other:ab
```

**Note:** The **user** entry for **denisf** and the **foreign_other** entry for **cella** both specify an empty permission set with notation - (dash), meaning that identities corresponding to these entries are explicitly denied all permissions. Also, the numbered lists in the discussions that follow correspond to stages 1, 2, 3, 4, 5 and 6 of the access-check algorithm described in “Access Checking” on page 440.

The principal **silviob** requests permission **a**. Assume that this principal’s privileges include membership in the group **projecty** and that they are not certified:

1. There is no **user_obj** entry, so this check can produce no permissions.
2. There is no **user** entry for this principal, so this check produces no permissions.
3. There is no entry for the group **projecty**, so this check yields no permissions.
4. There is no **other_obj** entry, so this check can produce no permissions.
5. The principal is local, so no **foreign_other** entry can be a match; this check produces no permissions.
6. Having failed to match any entry examined in the preceding checks, the principal matches the **any_other** entry, which produces the permission set **ab**. There is no **mask_obj** entry, but there is the **unauthenticated** mask entry, which specifies the permission set **a**. Applying the **unauthenticated** mask to this privilege attribute entry produces the effective permission **a**.

The permission requested (**a**) is a member of the effective permission set (**a**), so this principal’s request is granted.

## Name-Based Authorization

The Kerberos authentication service, on which the DCE Shared-Secret Authentication protocol is based, authenticates the string name representation of a principal. DCE Security converts these string representations to UUIDs. An ACL manager uses these UUIDs to make authorization decisions. However, because some existing (non-DCE) applications use Kerberos authentication, DCE Security supports an authorization protocol based on principal string names: Name-Based Authorization.

It is assumed that applications using Name-Based Authorization have a means to associate string names with permissions, because DCE Security offers no such facility. In Name-Based Authorization, there is no UUID representation of privilege attribute data, and DCE ACL managers recognize only UUIDs. Thus, if an application uses Name-Based Authorization, a principal’s privilege attributes are represented as an anonymous PAC. Such PAC data can only match the ACL entry types **other_obj**, **foreign_other**, or **any_other**, and are masked by the **unauthenticated** mask.

There is essentially no intercell security for an application that uses the Name-Based Authorization protocol. Such applications never communicate with the Privilege Service, that evaluates intercell trust.

# Chapter 26. GSSAPI Credentials

A GSSAPI *credential* is a data structure that provides proof of an application's claim to a principal name. An application uses a credential to establish its global identity. The global identity can be, but is not necessarily, related to the local user name under which the application (either the initiator or the acceptor) is running.

A credential can consist of either of the following:

- DCE Login context
- Principal name

There are three types of credentials, as shown in Table 22.

*Table 22. Credential Types*

| Credential | Content |
|---|---|
| INITIATE | A login context only. This credential identifies applications that only initiate security contexts. |
| ACCEPT | Principal name and an associated entry key table. This credential identifies applications that only accept security contexts. |
| BOTH | A login context and principal name with a key table entry. This credential identifies applications that can either initiate or accept security contexts. |

Credentials are maintained internally to GSSAPI. When they establish a security context, applications use credential handles to point to the credentials they need.

When an application initiates or accepts a security context, it can use GSSAPI routines with either a default credential or a specific credential handle. This chapter discusses how applications:

- Use default credentials
- Create credential handles to refer to specific credentials
- Delegate credentials

For detailed information on the GSSAPI routines, refer to the “DCE Generic Security Service API” chapter in the *z/OS DCE Application Development Reference*.

## Using Default Credentials

A **default credential** is a credential that is:

- Generated by either of the following routines:
  - **gss_init_sec_context()**
  - **gss_accept_sec_context()**
- Based on the following information:
  - The DCE default login context for the application (for INITIATE type credentials)
  - The registered principal name in the token (for ACCEPT or BOTH type credentials)

When an application calls the GSSAPI routine to either initiate (**gss_init_sec_context()**) or accept (**gss_accept_sec_context()**)a security context, it can specify the use of its default credential.

Use default credentials to help ensure the portability of your applications.

### Initiate a Security Context

To use a default credential when initiating a security context, an application calls the **gss_init_sec_context()** routine and specifies **GSS_C_NO_CREDENTIAL** as the input claimant credential handle to the routine. The routine uses the initiator's DCE default login context to generate the default credential. The credential is an INITIATE type credential.

You can change the default login context by calling the DCE sec_login routines. For information on the sec_login routines, see the *z/OS DCE Application Development Reference*.

### Accept a Security Context

To use a default credential when accepting a security context, an application calls the **gss_accept_security_context()** routine and specifies **GSS_C_NO_CREDENTIAL** as the verifier credential handle to the routine. The GSSAPI uses a principal name registered for the context acceptor to generate the default credential handle. The credential is an ACCEPT credential type.

## Creating New Credential Handles

An application can create a new credential handle to pass to the **gss_init_sec_context()** or **gss_accept_sec_context()** routines. An application might create a credential handle rather than use the default credential for the following reasons:

- Limit the identities the application can use
- Provide an additional identity for the application

### Initiating a Security Context with New Credential Handles

To create a credential handle for an INITIATE credential type, the application calls the **gssdce_login_context_to_cred()** routine and specifies its login context as input to the routine. The routine creates a credential handle that points to the credential consisting of that login context.

An application can also use a BOTH type credential to initiate a security context. Use the **gss_acquire_cred()** routine to create a BOTH type credential, as explained in the next section.

When the application uses a BOTH credential, the **gss_acquire_cred()** routine creates a login context from the key table information. Then, it uses the login context to create the a credential. For more details, see *z/OS DCE Application Development Reference*.

### Accepting a Security Context Using New Credential Handles

To create new credential handle for an ACCEPT or BOTH type credential, an application calls the **gss_acquire_cred()** routine.

The **gss_acquire_cred()** routine uses a principal name and its entry in the key table to generate the credential handle. If the principal name has not yet been registered (using **gssdce_register_acceptor_identity()** or the **rpc_server_register_auth_info()** routines), the **gss_acquire_cred()** routine automatically registers it.

## Delegating Credentials

In delegation, an initiator forwards its identity to an acceptor, so the acceptor can use the identity to act as an agent for the initiator. There are two forms of delegation:

- Impersonation delegation
- Traced delegation

## Initiating a Security Context to Delegate Credentials

An application indicates that it wants to delegate credentials when it calls **gss_init_sec_context()** routine and sets the **GSS_C_DELEG_FLAG** flag to true. Notes added to the initiator's login context can indicate the type of delegation used and any restrictions in effect (for traced delegation only). If no delegation notes are included with the login context and the **GSS_C_DELEG_FLAG** flag is set, impersonation delegation is used.

## Accepting a Security Context with Delegated Credentials

If the **GSS_C_DELEG_FLAG** flag has been set when the security context was initiated, the **gss_accept_sec_context()** routine will pass a credential to the acceptor. The routine:

1. Uses information from the input token to create the appropriate delegated credential
2. Creates an impersonation or traced delegation credential with an INITIATE credential type
3. Passes the delegated INITIATE credential to the acceptor

The principal named in the delegated INITIATE credential is the name of the initiator (for impersonation delegation) or the acceptor acting *for* the initiator (for traced delegation). The acceptor uses the credential to act for the initiator, initiating security contexts as appropriate.

# Chapter 27. The Extended Privilege Attribute Application Program Interface

This chapter describes the Extended Privilege Attribute (EPA) API. The EPA facility addresses the requirements of complex distributed systems by allowing clients and servers to invoke secure operations through one or more intermediate servers.

In a simple client/server distributed environment, most operations involve two principals: the initiator of the operation and the target of the operation. The target of the operation makes authorization decisions based on the identity of the initiator. However, in distributed object oriented environments, there is frequently a need for server principals to perform operations on behalf of a client principal. In these cases, it may not be enough for authorization decisions to be based simply on the identity of the initiator, since the initiator of the operation may not be the principal that requests the operation.

To handle these cases, the EPA API provides routines that allow principals to operate on objects on behalf of (as delegates of) an initiating principal. The collection of the delegation initiator and the intermediaries is referred to as a *delegation chain*. Using the EPA and related **sec_login** calls, an application may be written that allows client principal A to invoke an operation on server principal C via server principal B. The Security service will know the true initiator of the operation (principal A) and can distinguish the delegated operation from the same operation invoked directly by principal A.

The EPA interface consists of the Security credential calls (**sec_cred**) that extract privilege attributes and authorization data from an opaque binding handle to authenticated credentials. In addition, the following **sec_login** calls of the Login API are used to establish delegation chains and to perform other delegation related functions.

- **sec_login_become_initiator**
- **sec_login_become_delegate**
- **sec_login_become_impersonator**
- **sec_login_cred_get_delegate**
- **sec_login_cred_get_initiator**
- **sec_login_cred_initialize_cursor**
- **sec_login_disable_delegation**
- **sec_login_set_extended_attrs**

## Identities of Principals in Delegation

The identity of principals in a delegation chain is maintained in Extended Privilege Attribute Certificates (EPACs), as are the identities for all DCE principals. Each EPAC contains the name and group memberships of a principal in the delegation chain and any extended attributes that apply to the principal. The delegation chain includes an EPAC for each member of the delegation chain.

When delegation is in use, the target server receives the delegation chain, and thus knows the privilege attributes of the delegation chain initiator and each intermediary (delegate) in the chain. Authorization decisions can then be made based on the identities of all principals involved in the operation.

## ACL Entry Types for Delegation

When a server's ACL manager is presented with credentials to use as a base of an authorization decision, the manager evaluates the privilege attributes of each principal involved in the delegation chain. The ACL manager grants access for the requested operation only if all principals in the delegation chain have the necessary permissions on the object that is the eventual target of the operation.

For the initiator of the delegation chain, permission on the target object must be granted directly using any of the following standard ACL entry types:

- **user_obj**
- **user**
- **foreign_user**
- **group_obj**
- **group**
- **foreign_group**
- **foreign_other**
- **other_obj**
- **foreign_other**
- **any_other**
- **extended**

For intermediaries in a delegation chain, permissions to a target object can be granted directly to the intermediary with the standard ACL entry type described above or permissions can be granted by delegate ACL entries. Delegate ACL entries grant permissions to principals only if they are acting as delegates. The following delegate ACL entry types are available:

- **user_obj_delegate**
- **user_delegate**
- **foreign_user_delegate**
- **group_obj_delegate**
- **group_delegate**
- **foreign_group_delegate**
- **foreign_other_delegate**
- **other_obj_delegate**
- **foreign_other_delegate**
- **any_other_delegate**

Note that to perform an operation, all delegates in the chain must have the appropriate permissions. For example, assume a delegation chain consists of Principal A (the initiator) and Principal's B and C (the intermediaries). To perform the operation, the delegation chain requires **Mrw** permissions on Server X. One way of granting these permission is to grant them directly to each member of the delegation chain, as shown below:

```
user:Principal A:Mrw
user:Principal B:Mrw
user:Principal C:Mrw
```

Providing access directly also allows each intermediary in the chain to perform the operation of their own initiative, a consequence that may or may not be desired. To specify that Principals B and C may only be intermediaries operating on behalf of an authorized initiating principal without granting them the ability to perform the operation on their own, use delegation entries. In this case the Server X's ACL would contain the following entries:

```
user:Principal A:Mrw
user_delegate:Principal B:Mrw
user_delegate:Principal C:Mrw
```

### ACL Checking for Delegation

To determine permissions, the ACL manager first uses the standard access-check algorithm (described in Chapter 25, “Authorization” on page 435) to determine the permissions to grant to the delegation initiator. If the requested permission is not granted, access is denied.

If the requested permission is granted, the ACL manager then checks the permissions granted to the delegates in the chain. This checking is similar to the standard access-check algorithm, but it takes into account any additional delegate permissions granted to the delegates. If the requested permission is not granted to all delegates, access is denied. If the requested permission is granted to all delegates, access is granted.

## Calls to Establish Delegation Chains

The following **sec_login** API calls set up a delegation chain:

- **sec_login_become_initiator**

  Enables delegation for a client. The principal that executes this call is known as the delegation initiator.

- **sec_login_become_delegate** and **sec_login_become_impersonator**

  Causes an intermediate server to become a delegate in delegation chain. The principals that execute these calls are known as intermediaries in the delegation chain.

The **sec_login_become_delegate** call should be used if the traced delegation has been enabled. The **sec_login_become_impersonator** should be used if simple delegation has been enabled. See “Types of Delegation” for more information.

The following sections describe the information supplied to the calls that establish delegation chains.

### Types of Delegation

When a client application calls **sec_login_become_initiator** to enable delegation, that application specifies the type of delegation that should be enabled. The delegation type can be:

- **Traced Delegation**

  Includes the identities of all members of the delegation chain in the credentials used for authorization. To become an intermediary in a traced delegation chain, server principals use the **sec_login_become_delegate** call.

  Note that ACLs on objects that are targets of traced delegation, must grant the requested permission (or delegate permission) to each member of the delegation chain.

- **Impersonation**

Includes only the identity of the initiator of the delegation chain used for authorization. All intermediaries *impersonate* the delegation initiator. To become an impersonator principals use the **sec_login_become_impersonator** call.

Note that ACLs on objects that are targets of impersonation, need list only the delegation initiator, not each delegate in the chain.

Generally, traced delegation is the preferred method. The high degree of location transparency inherent in simple delegation greatly increases the risk of a client being compromised by a Trojan horse application.

When server principals run the **sec_login_become_delegate** or **sec_login_become_impersonator** call to become an intermediary in a delegation chain, they must also specify the delegation type as input to the call. The type they specify must be the same type as the delegation type specified by the initiator of the chain (unless they specify no delegation).

## Target and Delegate Restrictions

When a principal enables delegation or becomes an intermediary in a delegation chain, the principal may specify target and delegate restrictions. Target restrictions identify the server principals (by UUID) to which the identities in a delegation chain can be projected. Delegate restrictions identify the server principals that can further project the delegation chain.

If a target restriction prohibits a server from seeing an identity in a delegation chain, the Security runtime replaces that identity with the identity of the anonymous principal. If a delegate restriction prohibits a principal from being an intermediary in a chain, then the Security runtime replaces that principal's identity with the identity of the anonymous principal. This replacement with the anonymous identity allows the authenticated RPC call to complete. Whether the operation requested by the delegation chain is performed can be controlled by ACL entries that grant permission to the anonymous principal on the objects that are the targets of the delegated operation.

If no delegate restrictions are supplied, any principal can be an intermediary in the delegation chain. If any delegate restrictions are supplied, then only those supplied can further transmit the delegation chain.

**The Anonymous Principal:** The Security service replaces those identities in the delegation chain that are not allowed to be seen by target or delegate restrictions with the UUIDs associated with the anonymous principal's identity. These UUIDs are:

- Anonymous principal UUID: fad18d52-ac83-11cc-b72d-0800092784e9
- Anonymous group UUID: fc6ed07a-ac83-11cc-97af-0800092784e9

The **other_obj**, **any_other**,**other_obj_deleg**, and **any_other_deleg** ACL entries define the anonymous principal's access to objects. The entries must be set up just as for any other principal. The appropriate direct or delegate permissions must be granted to the anonymous principal or the delegated operation will fail.

**Target and Delegate Restriction Syntax:** Target and delegate restrictions are expressed as a list of values of type **sec_id_restriction_t**. This data type consists of a UUID and an entry type. The entry type specifies whether the UUID identifies a principal, a group, or any other principals (in a manner similar to the **any_other** ACL entry type). As in ACL entry types, the target restriction entry types can refer to principals and groups from the local cell or from foreign cells.

The possible delegation entry types are:

| | |
|---|---|
| **sec_rstr_e_type_user** | The target or delegate is a local principal identified by UUID. |

| | |
|---|---|
| **sec_rstr_e_type_group** | The target or delegate is any member of a local group identified by UUID. |
| **sec_rstr_e_type_foreign_user** | The target or delegate is a foreign principal identified by principal and cell UUID. |
| **sec_rstr_e_type_foreign_group** | The target or delegate is any member of a foreign group identified by group and cell UUID. |
| **sec_rstr_e_type_foreign_other** | The target or delegate is any principal that can authenticate to the foreign cell identified by UUID. |
| **sec_rstr_e_type_any_other** | The target or delegate is any principal that can authenticate to any cell. |
| **sec_rstr_e_type_no_other** | No principal can act as a target or delegate. |

## Optional and Required Restrictions

When a principal calls the **sec_login_become_initiator** to enable delegation or the **sec_login_become_delegate** or **sec_login_become_impersonator** to become an intermediary, the principal can specify optional and required restrictions. Optional and required restrictions are provided for use by applications that have specific authorization requirements. These restrictions, which are defined by the application, can be set by initiators or intermediaries, and are interpreted and enforced by application target servers. Servers can ignore optional restrictions that they cannot interpret, but they must reject requests associated with a required restriction that they cannot interpret. Both optional and required restrictions are supplied as values of type **sec_id_opt_req_t**. They are inserted in an EPAC by the Privilege Server and evaluated by the target server application.

## Compatibility between z/OS and Pre-OS/390® Servers and Clients

Prior to OS/390 DCE a principal's privilege attributes were stored in a Privilege Attribute Certificate (PAC). In z/OS the PAC is replaced by an Extended Attribute Certificate (EPAC), which includes the PAC and:

- Target, delegate, optional, and required restrictions.
- Extended registry attributes described in “Extended Registry Attribute API” on page 465.

Additionally, authorization credentials can consist of multiple EPACs, as in delegation chains, instead of a single PAC.

When a pre-OS/390 client interacts with a z/OS server or vice versa, the z/OS server requires an EPAC and the pre-OS/390 server requires a PAC.

For z/OS servers, the Security runtime automatically converts the PAC supplied by a pre-OS/390 client to an EPAC. For pre-OS/390 servers, the Security runtime automatically extracts PAC data from the credentials supplied by the z/OS client. However, because an EPAC for a delegation chain contains the privilege attributes of multiple principals and a PAC contains only one set of privilege attributes, the principals engaged in delegation must specify how to handle this issue of multiple versus. single identities.

When a principal initiates delegation or becomes an intermediary in a delegation chain, that principal can specify whether to use the privilege attributes of the chain initiator or the last intermediary in the chain to construct the PAC required by a pre-OS/390 server. This compatibility decision is specified as a value of type **sec_id_compatibility_mode_t**, which is set to one of the following three values:

| | |
|---|---|
| **sec_id_compat_mode_none** | Compatibility mode is off. The Security runtime supplies the application server with an unauthenticated PAC. |

| | |
|---|---|
| **sec_id_compat_mode_initiator** | Compatibility mode is on. The pre-OS/390 PAC data is extracted from the EPAC of the delegation initiator. |
| **sec_id_compat_mode_caller** | Compatibility mode is on. The pre-OS/390 PAC data extracted from the EPAC of the last intermediary in the delegation chain. |

## Calls to Extract Privilege Attribute Information

The EPA API **sec_cred** and and Login API **sec_login_cred** calls extract privilege attribute information. These calls return information associated with an opaque handle to an authenticated identity.

The **sec_cred** calls are used by servers that have been called by a client with authenticated credentials. Following are the calls and the information they return:

| | |
|---|---|
| **sec_cred_get_authz_session_info** | Returns a client's authorization information |
| **sec_cred_get_client_princ_name** | Returns the principal name of the client |
| **sec_cred_get_deleg_restrictions** | Returns delegate restrictions |
| **sec_cred_get_delegate** | Returns a credential handle to the privilege attributes of a delegate in a delegation chain |
| **sec_cred_get_delegation_type** | Returns the delegation type |
| **sec_cred_get_extended_attrs** | Returns extended attributes |
| **sec_cred_get_initiator** | Returns a credential handle to the privilege attributes of the initiator of a delegation chain |
| **sec_cred_get_opt_restrictions** | Returns optional restrictions |
| **sec_cred_get_pa_data** | Returns privilege attributes from a credential handle |
| **sec_cred_get_req_restrictions** | Returns required restrictions |
| **sec_cred_get_tgt_restrictions** | Returns target restrictions |
| **sec_cred_get_v1_pac** | Returns pre-OS/390 privilege attributes |
| **sec_cred_is_authenticated** | Returns true if the caller's privilege attributes are authenticated or a false otherwise |

The **sec_login_cred** calls are used by clients that are part of a delegation chain. Following are the calls and the information they return:

| | |
|---|---|
| **sec_login_cred_get_delegate** | Returns the privilege attributes of a delegate in a delegation chain. |
| **sec_login_cred_get_initiator** | Returns the privilege attributes of the initiator of a delegation chain. |

## Opaque Handles for sec_cred Calls

The **sec_cred** and **sec_login** calls discussed in this chapter return information about authenticated principals associated with an opaque credential handle supplied to the call. Two credential handles are used:

- **sec_login_handle_t** (returned by a client-side **sec_login_get_current_context** call).
- **rpc_authz_cred_handle_t** (returned by a server-side **rpc_inq_auth_caller** call).

These are handles to all the credentials in a delegation chain. The **sec_login_cred_get_initiator**, **sec_login_cred_get_delegate**, **sec_cred_get_initiator**, and **sec_cred_get_delegate** return a handle of type **sec_cred_pa_handle_t**, which is a handle to the extended privilege attributes of a particular identity in a delegation chain. The other **sec_cred** and **sec_login** calls discussed in this chapter take the **sec_cred_pa_handle_t** handle and return the requested information for the particular identity.

## Disabling Delegation

The Login API **sec_login_disable_delegation** call disables delegation for a specified login context. It returns a new login context of type **sec_login_handle_t** without any delegation information and prevents any further delegation.

## Setting Extended Attributes

The Login API, **sec_login_set_extended_attrs** call adds extended registry attributes to a login context. The extended registry attributes must have been established and attached to the object using the Extended Registry Attribute API. (For more information see “Extended Registry Attribute API” on page 465.)

# Chapter 28. The Registry Application Program Interface

This chapter describes the Registry API. Like the other Security APIs, this one provides a simpler binding mechanism than the standard RPC handle structure. It includes facilities for creating and maintaining the Registry database. Applications that run in the z/OS DCE DCE Registry environment (that is, those that assume the presence of the z/OS DCE Registry tools and servers) usually do not need to call this API.

## Binding to a Registry Site

Although it is often convenient to speak of the Registry database in a way that implies that it is a single physical database, the Registry database is replicated in all but the very smallest cells. Replication reduces network traffic and increases the availability of Registry data to clients. A cell's Registry database usually consists of an update site (also known as the master site) and a number of query sites (also known as read-only sites.). Changes to data at the master site are distributed to its query sites by messages sent by the master. Query sites can only satisfy requests for data (for example, **sec_rgy_acct_lookup()**, which returns account information). Requests for database changes (for example, **sec_rgy_acct_passwd()**, which changes the password for an account) must be directed to the master site; a query site that receives such a request returns an error.

To submit requests to the Registry Server, a client must first select a site and bind to it. The client can select a site by name, ask the Directory Service to bind to the master site, or select an arbitrary site. In addition, a client can select a cell and bind to a registry site in that cell.

The Registry API enables a client to communicate with the Registry Server using a specified authentication protocol at a specified protection level, using a specified authorization protocol. For instance, a developer can decide that the protection level for communicating with an update site should be higher (that is, more secure) than that for a query site. The developer can decide the relatively infrequent changes to Registry data should be done in a highly secure manner, and the authentication overhead should be reduced for the more frequent requests for Registry data. The Registry API accommodates these varying needs.

The following calls bind a client to a Registry Server in preparation for Registry operations. The argument list of these calls enables an application to specify the authentication protocol, the protection level, and the authorization protocol used.

| | |
|---|---|
| **sec_rgy_site_bind()** | Binds to a specified site. |
| **sec_rgy_site_bind_update()** | Binds to any update site. |
| **sec_rgy_site_bind_query()** | Binds to any query site. |
| **sec_rgy_cell_bind()** | Binds to any registry site in a specified cell. |
| **sec_rgy_site_binding_get_info()** | Extracts the Registry site name and security information from the binding handle. |

The following calls are similar to the binding calls described previously, except that an application cannot specify security information. However, by default, the following calls use DCE Shared-Secret Authentication, the packet-integrity level of protection, and DCE Authorization:

| | |
|---|---|
| **sec_rgy_site_open()** | Binds to the specified site. |
| **sec_rgy_site_open_update()** | Binds to any update site. |
| **sec_rgy_site_open_query()** | Binds to any query site. |
| **sec_rgy_site_get()** | Gets the Registry site name from the binding handle. |

The following calls provide miscellaneous binding management functionality:

| | |
|---|---|
| **sec_rgy_site_close()** | Ends binding to a Registry site and frees resources associated with this binding |
| **sec_rgy_site_is_readonly()** | Tests whether a bound site is an update or query site. |

## The Registry Database

The Registry database comprises three container objects:

| | |
|---|---|
| **principal** | Contains principal names; each name is associated with account information that is also specified here (for example, the name of the primary group). |
| **group** | Contains groups and the names of their member principals. |
| **organization** | Contains organizations and the names of their member principals. |

These three objects are referred to as name domains, and each member of a domain is referred to as a PGO item. Principal items are contained in the principal domain, groups in the group domain, and organizations in the organization domain. A principal might have a name such as **/rd/writers/tom,** from which you might infer that **tom** is a member of the group **writers** and the organization **rd**. However, this is not the case because the name **/rd/writers/tom** only indicates that **tom** and the data corresponding to the account of this principal (if any) reside in **/rd/writers** in the principal domain. There can also be a group named **/rd/writers** in the group domain, but the principal **tom** is not a member unless he is explicitly named in the group **/rd/writers** in the group domain.

Each PGO item consists of a printstring name, a UUID, and a UNIX number (for compatibility with UNIX system security interfaces). For various administrative reasons, it is frequently convenient to be able to refer to a PGO item by more than one name. Consequently, some PGO items are aliases for other items. An alias uses the same UUID and UNIX number as the PGO item to which it refers, but contains only a pointer to that item.

The Registry also contains the **rgy** object, which describes Registry properties and policies, and organization policies.

## Creating and Maintaining PGO Items

The PGO items in the Registry database are created and maintained with routines that are prefixed **sec_rgy_pgo**. The contents of a PGO item vary with the domain. If the domain is **group** or **organization**, the contents are the membership list of principal names. If the domain is **principal**, the contents are the data corresponding to the Registry account using that name.

The **sec_rgy_pgo()** interface contains the following calls for maintaining the PGO trees:

| | |
|---|---|
| **sec_rgy_pgo_add()** | Adds a PGO item. |
| **sec_rgy_pgo_delete()** | Deletes a PGO item. |
| **sec_rgy_pgo_rename()** | Changes the name of a PGO item. |
| **sec_rgy_pgo_replace()** | Replaces information corresponding to the specified PGO item. |

The **sec_rgy_pgo()** interface contains the following calls for maintaining PGO membership lists:

| | |
|---|---|
| **sec_rgy_pgo_add_member()** | Adds a member to a group or organization membership list. |
| **sec_rgy_pgo_delete_member()** | Deletes a member from a group or organization membership list. |

| | |
|---|---|
| **sec_rgy_pgo_get_members()** | Returns a list of members of a group or organization. |
| **sec_rgy_pgo_is_member()** | Tests whether a principal is a member of a specified group or organization. |

The **sec_rgy_pgo()** interface contains the following calls for retrieving PGO item data:

| | |
|---|---|
| **sec_rgy_pgo_get_by_id()** | Returns the PGO item with the specified UUID. |
| **sec_rgy_pgo_get_by_name()** | Returns the PGO item with the specified name. |
| **sec_rgy_pgo_get_by_unix_num()** | Returns the PGO item with the specified UNIX number. |
| **sec_rgy_pgo_get_next()** | Returns the PGO item that follows the last PGO item returned. |

The **sec_rgy_pgo()** interface also contains routines that convert PGO item specifiers, as follows:

| | |
|---|---|
| **sec_rgy_pgo_id_to_name()** | Returns the name for a PGO item identified by its UUID. |
| **sec_rgy_pgo_id_to_unix_num()** | Returns the UNIX number for a PGO item identified by its UUID |
| **sec_rgy_pgo_name_to_id()** | Returns the UUID for a named PGO item. |
| **sec_rgy_pgo_unix_num_to_id()** | Returns the UUID or a PGO item identified by its UNIX number. |
| **sec_rgy_pgo_name_to_unix_num()** | Returns the UNIX number for a PGO item identified by its name. |
| **sec_rgy_pgo_unix_num_to_name()** | Returns the name for a PGO item identified by its UNIX number. |

## Creating and Maintaining Accounts

The login-name field of a DCE account contains a principal name, a primary group name, and an organization name. The account can also contain a project list (also known as a concurrent group set) that specifies all the groups to which the principal corresponding to the account belongs. However, the login name field itself specifies only one group name.

An account can be added to the Registry database only when all of its constituent PGO items are established. For instance, to create an account with the principal name **tom,** the group name **writers**, and the organization name **rd**, all three names must exist as individual PGO items in the database. If the user principal, in this case **tom**, has the appropriate ACL permissions, its membership will be added automatically when it is created. Otherwise, the **writers** group and the **rd** organization must specify that **tom** is a member.

When an account is created with **sec_rgy_acct_add()** (and if a project list is enabled for the new account), the call scans the groups in the Registry and creates a project list containing all the groups in which the principal name appears. Subsequently, the project list can be changed with the **sec_rgy_pgo_add_member()** and **sec_rgy_pgo_delete_member()** calls.

The following calls create and maintain accounts:

| | |
|---|---|
| **sec_rgy_acct_add()** | Adds an account to an existing principal item. |
| **sec_rgy_acct_delete()** | Deletes an account, leaving the principal item. |
| **sec_rgy_acct_rename()** | Changes an account login name. |

The following calls return the information in an account:

| | |
|---|---|
| **sec_rgy_acct_get_projlist()** | Returns the project list for an account. |
| **sec_rgy_acct_lookup()** | Returns all the account data. |

The following calls change the information in an account:

| | |
|---|---|
| **sec_rgy_acct_passwd()** | Changes an account password. |
| **sec_rgy_acct_replace_all()** | Replaces all of an account's data. |
| **sec_rgy_acct_admin_replace()** | Replaces only the administrative account data. |
| **sec_rgy_acct_user_replace()** | Replaces only the account data that is accessible to the user of the account. |

## Registry Properties and Policies

This section outlines some Registry API parameters that affect the cell as a whole, and the routines that enable an application to retrieve and set the parameter values.

**Registry Properties:** Several Registry parameters and flags affect all accounts in the Registry. They include the following:

- The version number of the Registry software used to create and read the Registry.
- The name and UUID of the cell associated with the Registry, and whether the current Registry site is an update site or a query site.
- Minimum and default lifetimes for certificates of identity issued to principals.
- Bounds on the UNIX numbers used for principals, and whether the UUIDs of principals also contain embedded UNIX numbers.

The routines associated with this parameter set are:

| | |
|---|---|
| **sec_rgy_properties_get_info()** | Returns registry properties. |
| **sec_rgy_properties_set_info()** | Sets registry properties. |

**The Registry Authentication Policy:** Another set of parameters affecting all principals is the Registry authentication policy. This set only controls the maximum lifetime of certificates of identity, on first issue and renewal. Accounts also have authentication policies and the policy in effect for any principal is the most restrictive combination of the Registry policy and the policy for a principal's account. The associated routines are:

| | |
|---|---|
| **sec_rgy_auth_plcy_get_info()** | Returns the authentication policy for an account. |
| **sec_rgy_auth_plcy_get_effective()** | Returns the effective authentication policy for an account. |
| **sec_rgy_auth_plcy_set_info()** | Sets the authentication policy for an account. |

**Organization Policies:** Another parameter set controls the set of accounts of principals that are members of an organization. These parameters control the lifetime and length of passwords, as well as the set of characters from which passwords may be composed. This parameter set also specifies the default life span of accounts associated with the organization. The routines associated with this parameter set are:

| | |
|---|---|
| **sec_rgy_plcy_get_info()** | Returns the policy for an organization. |
| **sec_rgy_plcy_get_effective()** | Returns the effective policy for an organization. |
| **sec_rgy_plcy_set_info()** | Sets the policy for an organization. |

## Routines to Return UNIX Structures

The Registry API provides calls to obtain registry entries in a UNIX style structure. These APIs return account and group entries similar to the **getpwnam**, **getgrnam**, **getpwuid**, and **getgrid** UNIX library routines. These APIs, which can be called by corresponding UNIX library routines to ensure compatibility with UNIX programs, are:

| | |
|---|---|
| **sec_rgy_unix_getpwnam()** | Returns a UNIX style password entry for an account specified by name. |
| **sec_rgy_unix_getgrpnam()** | Returns a UNIX style group entry for an account associated with a specified group name. |
| **sec_rgy_unix_getpwuid()** | Returns a UNIX style password entry for an account specified by UNIX ID. |
| **sec_rgy_unix_getgrgid()** | Returns a UNIX style group entry for an account associated with a specified group ID. |

## Miscellaneous Registry Routines

The Registry API includes a few miscellaneous routines:

| | |
|---|---|
| **sec_rgy_login_get_info()** | Returns login information for the specified account. |
| **sec_rgy_login_get_effective()** | Applies local overrides (if such data is available) to Registry account information and returns information about which account information fields have been overridden. **Note:** Override capability is not supported in z/OS DCE. |
| **sec_rgy_wait_until_consistent()** | Blocks until all previous database updates have been distributed to all sites. This is useful for applications that first bind and write to an update site, and then bind to an arbitrary query site and depend on up-to-date information. |
| **sec_rgy_cursor_reset()** | Resets the database cursor to return the first suitable entry. |

# Chapter 29. The Extended Attribute Application Program Interfaces

This chapter describes the Extended Attribute APIs. There are two Extended Attribute APIs: the Extended Registry Attribute (ERA) interface to create attributes in the registry database and the DCE Attribute interface to create attributes in a database of your choice.

The Extended Registry Attribute interface (consisting of **sec_attr()** calls) provides facilities for extending the Registry database by creating, maintaining, and viewing attribute types and instances, and providing information to and receiving it from outside attribute servers known as attribute triggers. It is the preferred API for Security schema and attribute manipulations. Application servers that manage legacy Security attributes or provide third-party processing of attributes stored in the registry database can export and implement the **sec_attr()** interface. Trigger servers are accessed through the **sec_attr_trig()** interface by the Security client agent during certain **sec_rgy_attr()** calls. The Extended Registry Attribute interface uses the same binding mechanism as the Registry API, described in Chapter 28, “The Registry Application Program Interface” on page 459.

The DCE Attribute interface (consisting of **dce_attr_sch()** calls) is provided for schema and attribute manipulation of data repositories other than the registry. Although similar to the ERA interface, the functionality of the DCE Attribute interface is limited to creating schema entries (attribute types). The interface does not provide calls to create and manipulate attribute instances or to access trigger servers.

The chapter first describes the Extended Registry Attribute interface and then the DCE Attribute interface. Finally is describes macros and utilities provided for developers who use either Attribute API.

## Extended Registry Attribute API

The Registry is a repository for principal, group, organization, and account data. It stores the network privilege attributes used by the DCE and account data used by local operating systems. This local account data, however, is appropriate only for UNIX operating systems. The Extended Registry Attribute Facility provides a mechanism for extending the Registry schema to include data (attributes) required by or useful to operating systems other than UNIX.

The ERA API provides the ability to define attribute types and to attach attribute instances to registry objects. Registry objects are nodes in the Registry database, to which access is controlled by an ACL Manager type. The Registry objects are:

- Principal
- Group
- Organization
- Policy
- Directory
- Replist
- Attr_schema

All registry objects and their accompanying ACL Manager Type are described in the *z/OS DCE Administration Guide*.

The ERA API also provides a trigger interface that application servers use to integrate their attribute services with the Extended Registry Attribute services.

## Attribute Schema

The schema extensions are implemented in a single attribute schema that is essentially a catalog of schema entries, each of which defines the format and function of an attribute type. The schema can be dynamically updated to create, modify, or delete schema entries.

The attribute schema is identified by the name **xattrschema** under the security junction point (usually **/.:/sec**) in the CDS namespace. Access to the attribute schema (also called schema) is controlled by an ACL on the schema object. The schema is propagated from the master security server to replicas, like other Registry data. Since the attribute schema is local to a cell, it defines the types that can be used within the cell, but not outside the cell (unless the type is also defined in another cell).

## Attribute Types and Instances

Each attribute type definition in the schema consists of attribute type identifiers (UUID and name) and semantics that control the instances of attributes of this type. In this book, schema entry refers to the registry entry that defines an attribute type.

An attribute instance is an attribute that:

- Is attached to an object
- Has a value (as opposed to an attribute type, which has no values but simply defines the semantics to which attribute instances of that attribute type must adhere).

Attribute instances contain the UUID of their attribute type.

## Attribute Type Components

The **sec_attr_schema_entry_t** data type defines an attribute type. This data type contains attribute type identifiers and characteristics.

The identifiers of attribute types are a name and a UUID. Generally the name is used for interactive access and the UUID for programmatic access.

Attribute type characteristics describe the format and function of the attribute type and thus control the format and function of instances of that type. These characteristics, all specified in the **sec_attr_schema_entry_t** data type, are described in the following sections.

**Attribute Encoding:** Attribute encoding defines the legal encoding for instances of the attribute type. The encoding controls the format of the attribute instance values, such as whether the attribute value is an integer, string, a UUID, or a vector of UUIDs that define an attribute set.

Attribute encodings are specified in the **sec_attr_encoding_t** data type (fully described in the *z/OS DCE Application Development Reference*).

The possible encodings for attribute types are:

| | |
|---|---|
| **any** | The attribute instance value can be of any legal encoding type. |
| **void** | The attribute instance has no value. It is simply a marker that is either present or absent. |

|  |  |
|---|---|
| **printstring** | The attribute value is a printable IDL character string using the DCE Portable Character Set (PCS). |
| **printstring_array** | The attribute value is an array of printstrings. |
| **integer** | The attribute value is a signed 32 bit integer. |
| **bytes** | The attribute value is a string of bytes. The byte string is assumed to be a pickle or is otherwise a self describing type. |
| **confidential_bytes** | The attribute value is a string of encrypted bytes. This encrypted data can be passed over the network and is available to user-developed applications. |
| **i18n_data** | An internationalized string of bytes with a tag identifying the OSF registered codeset used to encode the data. |
| **uuid** | A DCE UUID. |

**attr_set**

The value is an attribute set, a vector of attribute type UUIDs used to associate multiple related attribute instances (members of the set). The vector contains the UUIDs of each member of the set. Attribute sets provide a flexible way to group related attributes on an object for easier search and retrieval.

The attribute type UUIDs referenced in an attribute set instance must correspond to existing attribute schema entries. Although the members specified in a set are generally expected to be attached to the object to which the set instance is attached, no checking is done to confirm that they are. Thus, it is possible to create an attribute set instance on an object before creating member attribute instances on that object. A query on such an attribute set returns all instances of member attributes that exist on the object along with a warning that some attribute types were missing.

Note that attribute sets cannot be nested: a member UUID of an attribute set can not itself identify an attribute set.

A query on an attribute set expands to a query per the set's members. In other words, an attribute lookup operation on an attribute set returns all attribute instances that are members of the set, not the set instance itself. (Certain operations, **sec_rgy_attr_set_lookup_by_id** and **sec_rgy_attr_lookup_by_name**, can retrieve attribute set instances.)

Updates to an attribute set (**sec_rgy_attr_update**) do not expand the update to its members, but apply only to the attribute set. Since the value carried by a set instance is a vector containing the UUIDs of the member attribute types, an update makes changes only to the set's members, not the values carried by those member attributes. Deletions of attribute sets delete only the set instance, not the member instances.

Since the attributes that are set members exist independently of the attribute set, they can be manipulated directly like any other attribute.

**binding**

The attribute value is a **sec_attr_binding_info_t** containing authentication, authorization and binding information suitable for communicating with a DCE server.

**ACL Manager Set:** An attribute type's ACL manager set specifies the ACL Manager type or types (by UUID) that control access to the object types to which attribute instances of this type can be attached. Attribute instances can be attached only to objects protected by the ACL Manager types in the schema entry. For example, suppose an ACL manager set for an attribute type named **MVSname** lists only the ACL Manager type for principals. Then, instances of the attribute type named **MVSname** can be attached only to principals and not any other registry objects.

Access to an attribute instance is controlled by the ACL on the object to which the attribute instance is attached and access control is implemented by the object's ACL Manager type. For example, access to an attribute named **MVSname** on the principal object named **delores** is controlled by the ACL on the **delores** object.

Do not confuse access to an attribute type definition (a schema entry) with access to an attribute instance. As described previously, access to a schema entry is controlled by the ACL on the **xattrschema** object. Access to an attribute instance is controlled by the ACL on the object to which the attribute instance is attached.

In addition to the ACL Manager types, the ACL manager set defines the permission bits need to query, update, test, and delete instances of the attribute type. These bits are used by the object's ACL Manager to determine rights to the object's attributes.

The ACL Manager types and permissions defined for the attribute type apply to all instances of the attribute type.

Note that the ACL Manager facility supports additional generic attribute type permissions (**O** through **Z** inclusive). Administrators can assign these permissions to attribute types of their choice. All uses of these additional permission bits are controlled by the cell's administrator. See the *z/OS DCE Administration Guide* for more information.

**Attribute Flags:** Following is a description of the attribute type flags set in a schema entry:

***Unique Flag:*** The unique flag specifies whether or not the value of each instance of an attribute type must be unique within the cell. For example, assume that an instance of attribute type A is attached to 25 principals in the cell. If the unique flag is set on, the value of the A attribute for each of those 25 principals must be different. If it is set off, then all 25 principals can share the same value for attribute A.

***Multi-Valued Flag:*** The multi-valued flag specifies whether or not instances of the attribute can be multi-valued. If an attribute is multi-valued, multiple instances of the same attribute type can be attached to a single Registry object. For example, if the multi-valued flag is set on, a single principal can have multiple instances of attribute type A. If the flag is set off, a single principal can have only one instance of attribute type A.

All instances of multi-valued attributes share the UUID (the UUID of their attribute type), but the values carried by the instances differ. Generally, to access all instances of a multi-valued attribute, you supply the attribute UUID. To access a specific instance of a multi-valued attribute, you supply the UUID and the value carried by that instance.

***Reserved Flag:*** The reserved flag indicates whether or not the attribute type can be deleted from the schema. Note that when an attribute type is deleted, all instances of the attribute type are deleted. If the reserved flag is set on, the entry cannot be deleted. If the reserved flag is set off, authorized principals can delete the schema entry.

***Apply_Defaults Flag:*** The use_defaults flag indicates whether or not default attributes should be returned when objects are queried by a client with the **sec_rgy_attr_get_effective** call. If apply-defaults flag is set on, defaults are applied. If it is set off, defaults are not supplied.

Defaults are determined in the following manner:

1. If the requested attribute exists on the principal, that attribute is returned. If it does not, the search continues.
2. The next step in the search depends on the type of object:

   **For principals with accounts:**

   a. The organization named in the principal's account is examined to see if an attribute of the requested type exists. If it does, it is returned and the search ends. If it does not, the search continues to the **policy** object as described in step 2b.

   b. The registry **policy** object is examined to see if an attribute of the requested type exits. If it does, it is returned. If it does not, a message indicating the no attribute of the type exists for the object is returned.

**For principals without accounts, for groups, and for organizations:** the registry **policy** object is examined to see if an attribute of the requested type exits. If it does, it is returned. If it does not, a message indicating the no attribute of the type exists for the object is returned.

**Intercell Action Field:** The intercell_action field of the schema entry specifies the action that should be taken by the Privilege server when reading attributes from a foreign cell. This field can contain one of three values:

| | |
|---|---|
| **sec_attr_intercell_act_accept** | To accept the foreign attribute instance. |
| **sec_attr_intercell_act_reject** | To reject the foreign attribute instance. |
| **sec_attr_intercell_act_evaluate** | To call a remote trigger server to determine how the attribute instance should be handled. |

When the Privilege server generates a PTGT for a foreign principal, it retrieves the list of attributes from the foreign principal's EPAC.

These attributes instances may be attached to the principal object itself or attached to the group or organization object associated with the principal object.

The Privilege server then checks the local attribute schema for attribute types with UUIDs that match the UUIDs of the attribute instances from the foreign cell that are contained in the EPAC. At this point the Privilege Server takes one of two actions:

1. If the Privilege server cannot find a matching attribute type in the local attribute schema, it checks the **unknown_intercell_action** attribute on the **policy** object. If the **unknown_intercell_action** attribute is set to:

   - **sec_attr_intercell_act_accept**, the foreign attribute instance is retained and included in the EPAC generated for the object by the Privilege server.
   - **sec_attr_intercell_act_reject**, the foreign attribute is discarded

**Note:** The **unknown_intercell_action** attribute must be created by the system administrator and attached to the **policy** object. The attribute type, which takes the same values as the **intercell_action field**, has the following characteristics:

- Name: unknown_intercell_action
- Attribute UUID: 71e0ef2c-d12e-11cc-bb7b-080009353559
- Encoding: sec_attr_encoding_integer
- ACL Manager set: policy_acl_mgr
- Unique: false
- Multi-valued: false
- Reserved: true
- Comment text: Flag indicating whether to accept or reject foreign attributes for which no schema entry exists

2. If the Privilege server finds a matching attribute type in the local attribute schema, it retrieves the attribute. The action it takes depends on the setting of the attribute type's intercell action field and unique flag as follows:

   - If the intercell action field is set to **sec_attr_intercell_act_accept** and:
     - The unique flag is not set on, the Privilege server includes the foreign attribute instance in the principal's EPAC
     - The unique flag is set on, the Privilege server includes the foreign attribute instance in the principal's EPAC only if the attribute instance value is unique among all instances of the attribute type within the local cell

     **Note:** If unique attribute type flag is set on and a query trigger exists for a given attribute type, the intercell action field cannot be set to **sec_attr_intercell_act_accept** because, in this case, only the query trigger server can reasonably perform a uniqueness check.
   - If the intercell action field is set to **sec_attr_intercell_act_reject**, the Privilege server unconditionally discards the foreign attribute instance
   - If the intercell action field is set to **sec_attr_intercell_act_evaluate**, the Privilege server makes a remote **sec_attr_trig_intercell_avail** call to an attribute trigger using the binding information in the local attribute type schema entry. The remote attribute trigger decides whether to retain, discard, or map the attribute instance to another value(s). The Privilege server includes the values returned by the attribute trigger in the **sec_attr_trig_query** call output array in the principals EPAC.

**Attribute Scope:** The scope field controls the objects to which the attribute can be attached. If scope is defined, the attribute can be attached only to objects defined by the scope. For example, if the scope for a given attribute type is defined as the directory name **/principal/krbgt**, instances of that attribute type can be attached only to objects in the **/principal/krbgt** directory (a directory that by convention contains only cell principals). If the scope is narrowed by fully specifying an object in the **/principal/krbgt** directory. for example **/principal/krbgt/dresden.com**, then the attribute can be attached only to the **dresden.com** principal.

**Trigger Type Flag:** The schema entry trigger type flag specifies whether the trigger server associated with the attribute type is invoked for update or query operations. See "The Attribute Trigger Facility" on page 481 for more information on attribute triggers.

**Trigger Binding:** The schema entry trigger binding field contains a binding handle to a remote trigger that will perform processing for the attribute instances. See "The Attribute Trigger Facility" on page 481 for more information on attribute triggers.

## Calls to Manipulate Schema Entries

This section first introduces the **sec_attr_schema_entry_t** data type used by the calls that create and update schema entries that define attribute types. It then describes the calls that create, modify, delete, and read schema entries.

### The sec_attr_schema_entry_t Data Type

The **sec_attr_schema_entry_t** data type is used in the calls that create and update schema entries. The data type consists of four values and six other data types. The values used by the **sec_attr_schema_entry_t** are the attribute type name, UUID, scope, and a text field for comments.

The data types used by the **sec_attr_schema_entry_t** are:

| | |
|---|---|
| **sec_attr_sch_entry_flags_t** | That specifies the unique, multi-valued, reserved, and use_defaults attribute flags. |
| **sec_attr_acl_mgr_info_set_t** | That specifies the attribute type's ACL manager(s). This data type defines the attribute type ACL manager set. This data type contains an array of pointers of type **sec_attr_mgr_info_p_t**, which reference **sec_attr_acl_mgr_info_t** data types. There is one **sec_attr_acl_mgr_info_t** data type for each ACL manager associated with the attribute type. Each **sec_attr_acl_mgr_info_t** defines ACL manager UUID and the permission bits. |
| **sec_attr_encoding_t** | That specifies the schema entry encoding. |
| **sec_attr_trig_type_t** | That specifies the type of attribute trigger associated with the attribute type (if an attribute trigger is to be associated with the attribute type). See "The Attribute Trigger Facility" on page 481 for more information on attribute triggers. |
| **sec_attr_intercell_action_t** | That specifies the action to be taken attribute instances of this type that come from a foreign cell. |
| **sec_attr_bind_info_t** | That specifies binding information for the trigger server associated with the attribute type (if an attribute trigger is associated with the attribute type). The **sec_attr_bind_info_t** data type uses two other data types: **sec_attr_bind_auth_info_t** and **sec_attr_binding_t**. The **sec_attr_bind_info_t** structure for trigger binding is described fully in "The Attribute Trigger Facility" on page 481. |

Figure 100 on page 472 illustrates the structure of a **sec_attr_schema_entry_t** data type.

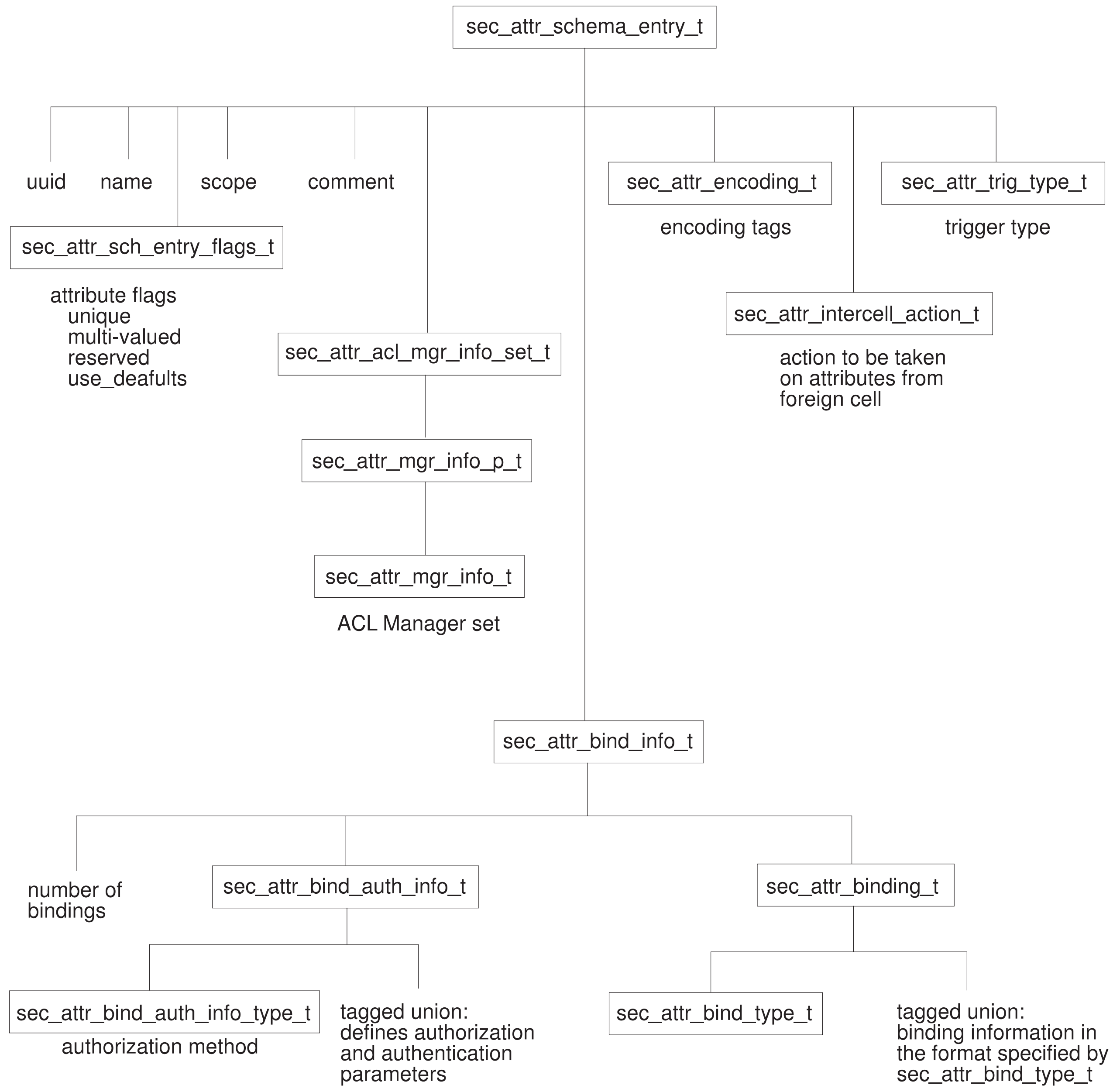

sec_attr_schema_entry_t
uuid
name
scope
comment
sec_attr_sch_entry_flags_t
attribute flags
unique
multi-valued
reserved
use_deafults
sec_attr_acl_mgr_info_set_t
sec_attr_mgr_info_p_t
sec_attr_mgr_info_t
ACL Manager set
sec_attr_encoding_t
encoding tags
sec_attr_trig_type_t
trigger type
sec_attr_intercell_action_t
action to be taken on attributes from foreign cell
sec_attr_bind_info_t
number of bindings
sec_attr_bind_auth_info_t
sec_attr_bind_auth_info_type_t
authorization method
tagged union: defines authorization and authentication parameters
sec_attr_binding_t
sec_attr_bind_type_t
tagged union: binding information in the format specified by sec_attr_bind_type_t

*Figure 100. The sec_attr_schema_entry_t Data Type*

## Creating and Managing Schema Entries

This section describes the calls to create, modify, and delete the schema entries that define attribute types.

**sec_rgy_attr_sch_create_entry():** The **sec_rgy_attr_sch_create_entry()** call creates a schema entry that defines an attribute type in the attribute schema.

This call uses the **sec_attr_schema_entry_t** data type that completely defines the schema entry, including:

- The attribute type name (generally used for interactive access) and UUID (generally used for programmatic access). Note that attribute instances share the name and UUID of their attribute type.
- The attribute's encoding (described in “Attribute Type Components” on page 466). The encoding is specified an enumerator of type **sec_attr_encoding_t**. For some kinds of encodings, additional data types are used to further specify the encoding information. These additional data types, the kinds of encodings that require them, and the purpose of the data types are listed in Table 23.

*Table 23. Encodings and Required Data Types*

| Encoding | Required Data Type | Purpose of Data Type |
|---|---|---|
| **sec_attr_enc_bytes** | **sec_attr_enc_bytes_t** | Defines the length of attribute values |
| **sec_attr_enc_confidential_bytes** | **sec_attr_enc_bytes_t** | Defines the length of attribute values |
| **sec_attr_enc_i18n_data** | **sec_attr_i18n_data_t** | Defines the i18n codeset |
| **sec_attr_enc_attr_set** | **sec_attr_enc_attr_set_t** | Defines the total number of members in the attribute set and the UUID of each member |
| **sec_attr_enc_printstring** | **sec_attr_enc_printstring_t** | Defines a single printstring |
| **sec_attr_enc_printstring_array** | **sec_attr_enc_str_array_t** | Defines an array of printstrings |

**sec_rgy_attr_sch_update_entry():** The **sec_rgy_attr_sch_update_entry()** call updates a schema entry that defines an attribute type.

The schema entry components that can be modified are controlled by the Extended Registry Attribute API and the *modify_parts* parameter of the **sec_rgy_attr_sch_update_entry()** call.

To ensure that Registry and access control data remains consistent, the Extended Registry Attribute API allows only the following schema entry components to be modified:

- Attribute name
- Reserved flag
- Apply defaults flag
- Intercell action flag
- Trigger binding
- Comment

Note that ACL Managers can be added to a schema entry's ACL Manager set, but they cannot be modified or deleted.

To modify any other schema entry fields implies a drastic change to the attribute type. If this change must be made, the schema entry must be deleted (which deletes all attribute instances of that type), and then re-created.

The *modify_parts* parameter of the **sec_rgy_attr_sch_update_entry()** call can also be used to prohibit modification of additional schema entry fields. This parameter, which is actually a

**sec_attr_schema_entry_parts_t** data type, lists the fields that can be modified by the call. Only those fields listed in **sec_attr_schema_entry_parts_t** can be modified.

The new values used to update the attribute type are supplied in a **sec_attr_schema_entry_t** data type.

**sec_rgy_attr_sch_delete_entry():** The **sec_rgy_attr_sch_delete_entry()** call deletes attributes types from the attribute schema. The attribute type to be deleted is specified by UUID. When an attribute type is deleted, all instances of that attribute type are invalidated.

## Reading Schema Entries

This section describes the calls that read schema entries and the cursor used by the **sec_rgy_attr_sch_scan()** call.

**Using sec_attr_cursor_t with sec_rgy_attr_sch_scan():** The **sec_rgy_attr_sch_scan()** call, which reads a specified number of attribute type entries from the attribute schema, uses a cursor of type **sec_attr_cursor_t**. This cursor must be allocated before it can be used as input to the **sec_rgy_attr_sch_scan()** call. In addition, it can also be initialized to the first attribute type entry in the schema, although this is not required. After use, the resources allocated to the **sec_attr_cursor_t** must be released.

The following calls allocate, initialize, and release a **sec_attr_cursor_t** for use with the **sec_rgy_attr_sch_scan()** call:

- **sec_rgy_attr_sch_cursor_init()**

  The **sec_rgy_attr_sch_cursor_init()** call allocates resources to the cursor and initializes the cursor to the first attribute type entry in the attribute schema. This call also supplies the total number of entries in the attribute schema as part of its output. The cursor allocation is a local operation. The cursor initialization is a remote operation and makes a remote call to the Registry.

- **sec_rgy_attr_sch_cursor_alloc()**

  The **sec_rgy_attr_sch_cursor_alloc()** call allocates resources to the cursor, but does not initialize the cursor. However, since the **sec_rgy_attr_sch_scan()** call will initialize the cursor if it is passed in uninitialized, you may prefer this call to limit the number of remote calls performed by an application. Be aware that the **sec_rgy_attr_sch_cursor_init()** call provides the total number of entries in the named schema, a piece of information not provided by the **sec_rgy_attr_sch_cursor_alloc()** call.

- **sec_rgy_attr_sch_cursor_release()**

  The **sec_rgy_attr_sch_cursor_release()** call releases all resources allocated to a **sec_attr_cursor_t** cursor used with the **sec_rgy_attr_sch_scan()** call.

- **sec_rgy_attr_sch_cursor_reset()**

  The **sec_rgy_attr_sch_cursor_reset()** call initializes a **sec_attr_cursor_t** cursor used with the **sec_rgy_attr_sch_scan()** call. The reset cursor can then be used without releasing and re-allocating.

**sec_rgy_attr_sch_scan():** The **sec_rgy_attr_sch_scan()** call reads a specified number of schema entries from the attribute schema.

The number of entries to read is specified as an unsigned 32-bit integer. The read begins at the entry at which the **sec_attr_cursor_t** cursor is positioned and continues through the number of entries specified. The cursor must be allocated but can be initialized or uninitialized, since **sec_rgy_attr_sch_scan()** initializes any uninitialized cursor it receives as input.

The call output includes an array of **sec_attr_schema_entry_t** values and a 32-bit integer that specifies the number of schema entries returned.

To read through all entries in a schema, continue making **sec_rgy_attr_sch_scan()** calls, until the **no_more_entries** message is received. When all calls are complete, release the resources allocated to the **sec_attr_cursor_t** cursor by using the **sec_rgy_attr_sch_cursor_release()** call.

**sec_rgy_attr_sch_lookup_by_id() and sec_rgy_attr_sch_lookup_by_name():** The **sec_rgy_attr_sch_lookup_by_id()** call reads the attribute schema entry identified by UUID. The output of the call is a **sec_attr_schema_entry_t** that contains the specified attribute type's name, UUID, and characteristics. Generally, this call is used for programmatic access.

For interactive access, use the **sec_rgy_attr_sch_lookup_by_name()** call. This call returns the same information as the **sec_rgy_attr_sch_lookup_by_id()** call, but specifies the schema entry to read by name instead of UUID.

### Reading the ACL Manager Types

Two calls retrieve the ACL Manager types that protect objects dominated by a named schema:

- **sec_rgy_attr_sch_get_acl_mgrs()**

  Retrieves the UUIDs of the ACL manager types protecting all objects in a named schema.

- **sec_rgy_attr_sch_aclmgr_strings()**

  Retrieves printable strings for each ACL manager type protecting objects in a named schema. The strings contain the ACL manager type's name, associated help information, and supported permission bits.

## Calls to Manipulate Attribute Instances

This section first introduces the **sec_attr_schema_t** data type used by the calls that create and update attribute instances. It then describes the calls that create, modify, delete, and read attribute instances. For all these calls, the object whose attributes should be accessed is identified by name and by the domain in which the object exists. (Note that the domain parameter is ignored for the **Policy** and the **Replist** objects.) Registry domains are described in Chapter 28, "The Registry Application Program Interface" on page 459.

## The sec_attr_t Data Type

The **sec_attr_t** data type is used in the calls that create and update attribute instances. The data type consists of a value of type **uuid_t** that identifies the attribute to be accessed by UUID and data type of **sec_attr_value_t**. The **sec_attr_value_t** data type is a tagged union of the actual value assigned (or to be assigned to the attribute instance) and a data type of **sec_attr_encoding_t** that specifies the encoding tags that define the attribute type characteristics. Figure 101 illustrates the structure of a **sec_attr_t** data type.

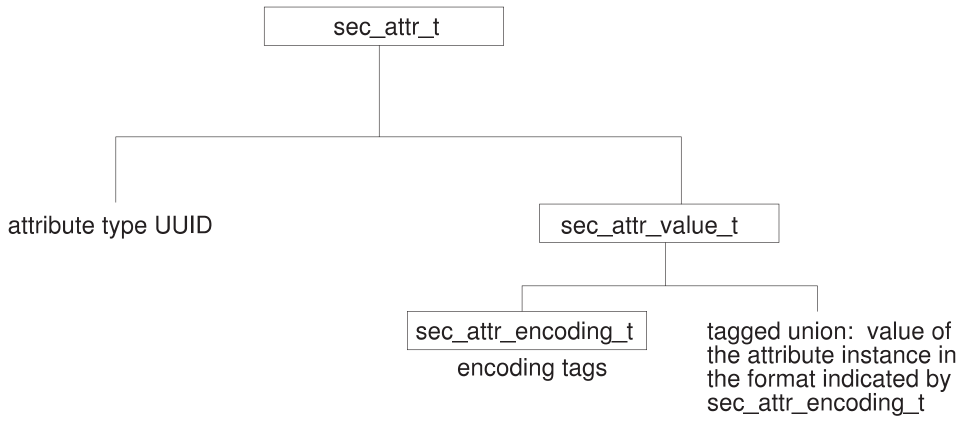

sec_attr_t
attribute type UUID
sec_attr_value_t
sec_attr_encoding_t
encoding tags
tagged union: value of the attribute instance in the format indicated by sec_attr_encoding_t

*Figure 101. The sec_attr_t Data Type*

## Creating and Managing Attribute Instances

This section describes the calls to create, modify, and delete the attribute instances.

### sec_rgy_attr_update():

The **sec_rgy_attr_update()** call creates new attribute instances and updates existing attribute instances attached to a object specified by name and Registry domain. The instances to be created or updated are passed as an array of **sec_attr_t** data types.

Because the new values are passed in as an array, if the update of any attribute instance in the array fails, all fail. However, to help pinpoint the cause of the failure, the call identifies the first attribute whose update failed in a failure index by array element number.

For existing attribute instances attached to object, the values passed in the array overwrite the existing values. In other words, if the UUID passed in the input array matches the UUID of an existing instance, the values passed in overwrite the existing values.

If the attribute instance does not exist, it is created. In other words, if the UUID passed in in the array does not match any other attribute type UUID attached to the object, a new attribute instance is created.

For multi-valued attributes, because every instance of the multi-valued attribute is identified by the same UUID, every instance is overwritten with the supplied value. For example, suppose object **delores** has three attributes of the multivalued type **security_role**. If you pass in one value for **security_role**, the values of all three are changed to the one you enter.

To change only one of the **security_role** values, you must supply the values that should be unchanged as well as the new value. For example, suppose object **delores** has three **security_role** attributes with values of level1, level2, and level3. To change level1 to level1.5 and retain level2 and level3, the input array must contain level1.5, level2, and level3.

To create instances of multi-valued attributes, you must create individual **sec_attr_t** data types to define each multi-valued attribute instance and then pass all of them in in the **sec_rgy_attr_update()** input array.

If an input attribute is associated with an update attribute trigger, the attribute trigger is invoked (by the **sec_attr_trig_update()** call) and the values in the **sec_rgy_attr_update()** input array are used as input to the update attribute trigger. The output values from the update attribute trigger are stored in the registry database and returned in the **sec_rgy_attr_update()** output array.

**sec_rgy_attr_test_and_update():** The **sec_rgy_attr_test_and_update()** call, like the **sec_rgy_attr_update()** call, creates new attribute instances and updates existing attribute instances attached to a object specified by name and Registry domain. However, it performs the update only if a set of specified attribute instances match the attribute instances that already exist for the object. This call is useful to ensure that updates are made only if certain conditions exist.

The attribute instances to be matched are passed in an input array of **sec_attr_t** values. Other than this conditional test, this call functions exactly the same as the **sec_rgy_attr_update()** call.

**sec_rgy_attr_delete():** The **sec_rgy_attr_delete()** call deletes the specified attribute instances from an object identified by name and Registry domain. The attribute instances to be deleted are passed in as an array of values of **sec_attr_t**.

To delete attribute instances that are not multi-valued and to delete all instances of a multi-valued attribute, an attribute UUID is all that is required. For these attribute instances, supply the attribute UUID in the input array and set the attribute encoding (in **sec_attr_encoding_t**) to **sec_attr_enc_void**.

To delete a specific instance of a multi-valued attribute, you must supply the UUID and value that uniquely identify the multi-valued attribute instance in the input array.

Note that if the deletion of any attribute instance in the array fails, all fail. However, to help pinpoint the cause of the failure, the call identifies the first attribute whose deletion failed in a failure index by array element number.

## Reading Attribute Instances

This section describes the calls that read attribute instances, and it describes the cursor used by the **sec_rgy_attr_lookup_by_id()** call.

**Using sec_rgy_attr_cursor_t with sec_rgy_attr_lookup_by_id():** The **sec_rgy_attr_lookup_by_id()** call, which reads attributes for a specified object, uses a cursor of type **sec_attr_cursor_t**. This cursor must be allocated before it can be used as input to the **sec_rgy_attr_lookup_by_id()** call. In addition, it can also be initialized to the first attribute in the specified object's list of attributes, although this is not required. After use, the resources allocated to the **sec_attr_cursor_t** must be released.

The following calls allocate, initialize, and release a **sec_attr_cursor_t** for use with the **sec_rgy_attr_lookup_by_id()** call:

- **sec_rgy_attr_cursor_init()**

  The **sec_rgy_attr_sch_cursor_init()** call allocates resources to and initializes the cursor to the first attribute in the specified object's list of attributes. This call also supplies the total number of attributes attached to the object as part of its output. The cursor allocation is a local operation. The cursor initialization is a remote operation and makes a remote call to the Registry.

- **sec_rgy_attr_cursor_alloc()**

The **sec_rgy_attr_cursor_alloc()** call allocates resources to the cursor, but does not initialize the cursor. However, since the **sec_rgy_attr_lookup_by_id()** call will initialize the cursor if it is passed in uninitialized, you may prefer this call to limit the number of remote calls performed by the application. Be aware that the **sec_rgy_attr_cursor_init()** call provides the total number of attributes attached to the specified object, a piece of information not provided by this call.

- **sec_rgy_attr_cursor_release()**

  The **sec_rgy_attr_cursor_release()** call releases all resources allocated to a **sec_attr_cursor_t** cursor used with the **sec_rgy_attr_lookup_by_id()** call.

- **sec_rgy_attr_cursor_reset()**

  The **sec_rgy_attr_cursor_reset()** call reinitializes a **sec_attr_cursor_t** cursor used with the **sec_rgy_attr_lookup_by_id()** call. The reset cursor can then be used without releasing and re-allocating.

**sec_rgy_attr_lookup_by_id():** The **sec_rgy_attr_lookup_by_id()** call reads attributes specified by UUID for an object specified by name and domain. Specifically the call returns:

- An array of **sec_attr_t** values.
- A count of the total number of attribute instances returned.
- A count of the total number of attribute instances that could not be returned because of size constraints of the **sec_attr_t** array. (Note that the call allows the size of the array to be specified.)

For multi-valued attributes, the call returns a **sec_attr_t** for each value as an individual attribute instance. For attribute sets, the call returns a **sec_attr_t** for each member of the set, but not the set instance. This routine is useful for programmatic access.

If the attribute instance to be read is not associated with a query trigger or no additional information is required by the query trigger, an attribute UUID is all that is required. For these attribute instances, supply the attribute UUID in the input array and set the attribute encoding (in **sec_attr_encoding_t**) to **sec_attr_enc_void**.

If the attribute instance to be read is associated with a query attribute trigger that requires additional information before it can process the query request, use a **sec_attr_value_t** to supply the requested information. To do this:

- Set the **sec_attr_encoding_t** to an encoding type that is compatible with the information required by the query attribute trigger.
- Set the **sec_attr_value_t** to hold the required information.

You can define the number of elements in the input array of **sec_attr_t** values (in the *num_attr_keys* parameter). If you define the number of elements as zero, the call returns all of the object's attribute instances that the caller is authorized to see. You should be aware, however, that if you define the number of elements as zero and the attribute is associated with a query attribute trigger, you will be unable to pass any information to the query attribute trigger.

**sec_rgy_attr_set_lookup_by_id():** The **sec_rgy_attr_set_lookup_by_id()** call reads attribute sets specified by set instance UUID for an object specified by name and domain. Specifically the call returns:

- A **sec_attr_t** for each attribute instance in the attribute set.
- A count of the total number of attribute set instances returned.
- A count of the total number that could not be returned because of size constraints of the **sec_attr_t** array. (Note that the call allows the size and length of the array to be specified.)

**Note:** Since attribute triggers can not be associated with an attribute set instance, this call provides no way to supply input data to a query attribute trigger.

**sec_rgy_attr_lookup_by_name():** The **sec_rgy_attr_lookup_by_name**() call reads a single attribute instance specified by name for an object specified by name and domain. The call returns a **sec_attr_t** for the specified attribute instance.

For multi-valued attributes, the call returns the first instance of the multi-valued attribute. (To retrieve every instance of a multi-valued attribute, use the **sec_rgy_attr_lookup_by_id** call.)

For attribute sets, the call returns the attribute set instance, not the member instances. To retrieve all members of the set, use the **sec_rgy_attr_lookup_by_id** call.

**Note:** This call provides no way to supply input data to a query attribute trigger. If the attribute to be read is associated with a query trigger that requires input data, use the **sec_rgy_attr_lookup_by_id** call.

## Searching for Attribute Instances

This section describes calls that search for objects that possess specified attribute instances.

**Using sec_attr_srch_cursor_t with sec_rgy_attr_srch Calls:** The **sec_rgy_attr_srch_names()** and **sec_rgy_attr_srch_names_attrs()** calls use a cursor of type **sec_attr_srch_cursor_t** initialized to a list of objects that meet a specified criteria. This cursor must be initialized before it can be used as input to the calls. After use, the resources allocated to the **sec_attr_srch_cursor_t** must be released. The following calls allocate, initialize, and release a **sec_attr_srch_cursor_t**.

- **sec_rgy_attr_srch_cursor_init()**

  The **sec_rgy_attr_srch_cursor_init()** call allocates resources to a **sec_attr_srch_cursor_t** and initializes the cursor to a search set of objects that possess specified attribute types or instances. The attribute types and attribute instance values searched for are identified in an array of **sec_attr_t** values.

  To find all attributes of a specified type, an attribute UUID is all that is required. For these attributes, supply the UUID of the desired attribute type in the input array and set attribute encoding (in **sec_attr_encoding_t**) to **sec_attr_enc_void**.

  To find a specific attribute instance, supply that attribute instance's UUID and value in the **sec_attr_t**. This method is useful for finding a single instance of a multi-valued attribute.

  If you specify an attribute set in the input array, only the set instance is found, not the members of the set.

  Query triggers are not supported during evaluation of the input array. Although an attribute type associated with a query trigger can be used in the array, if the attribute type also carries a value, a match will not be found. The match will fail because the query trigger is not executed; therefore, a value will not be retrieved for comparison against the search value.

You can define the number of elements in the input array (in the *num_search_attrs* parameter). If you define the number as zero, the search set is a list of all attributes in the named domain that are managed by the server to which the call is bound. The cursor is positioned at the beginning of the list.

- **sec_rgy_attr_srch_cursor_release()**

  The **sec_rgy_attr_srch_cursor_release()** call releases all resources allocated to a **sec_rgy_attr_srch_cursor** cursor.

### sec_rgy_attr_srch_names():

The **sec_rgy_attr_srch_names()** call returns the names of the objects that possess the attributes selected in the search set selected by the **sec_rgy_attr_srch_cursor_init()** call. Specifically this call returns:

- An array of names (**sec_rgy_name_t**) of the objects that possess the attributes specified as input to the **sec_rgy_attr_srch_cursor_init()** call.
- A count of the total object names returned.
- An array of values that specify the total number of attributes attached to each object whose name is returned in the names array.

The position of the element in the array of names and the number of attributes array connects the object name to the object's total number of attributes. The object named in position 1 of the array of names has the total number of attributes listed in position 1 total number of attributes array.

Before this call is executed, must execute the **sec_rgy_attr_srch_cursor_init()** call to select the search set, initialize a **sec_rgy_attr_srch_cursor_t** cursor to the beginning of the search set, and allocate resources to the cursor. After a **sec_rgy_attr_srch_names()** call is executed, you must execute a **sec_rgy_attr_srch_cursor_release()** call to release resources allocated to the cursor.

### sec_rgy_attr_srch_names_attrs():

The **sec_rgy_attr_srch_names_attrs()** call returns:

- The names of the objects that possess the attributes in the search set selected by the **sec_rgy_attr_srch_cursor_init()** call.
- For those objects, the attributes specified in an array of attribute UUIDs supplied as a parameter to this call.

Except for the fact that you can return attributes in addition to object names, the call is similar to the **sec_rgy_attr_srch_names_attrs()** call. Like the **sec_rgy_attr_srch_names()** call, this call returns an array of object names (**sec_rgy_name_t**), a count of the total object names returned, and array of values that specify the total number of attributes attached to each object whose name is returned in the names array. In addition, this call returns the following information that is not returned by the **sec_rgy_attr_srch_names()** call:

- A array of specified attribute UUIDs and values (**sec_attr_t**).
- A count of the total number of attributes that are attached to the object.
- A count of the total number of attributes returned.

The attributes to be returned are specified in an array type **uuid_t** values.

## The Attribute Trigger Facility

Some attribute types require the support of an outside server either to verify input attribute values or to supply output attribute values when those values are stored in an external database. Such a server could, for example, connect a legacy registry system to the DCE Registry. The attribute trigger facility provides for automatic calls to outside DCE servers, known as attribute triggers.

Trigger servers, which are written by application developers, export the **sec_attr_trig** interface. They are invoked automatically when an attribute that has been associated with an attribute trigger (during schema entry creation) is queried or updated. The attribute trigger facility consists of three components:

- The attribute schema trigger fields (**trig_types** and **trig_binding**) that associate an attribute trigger and its binding information with an attribute type. These fields are part of the standard creation of a schema entry that defines an attribute type. See “Attribute Schema” on page 466.
- The **sec_attr_trig** APIs that define the query and update trigger operations. The APIs are provided in the **sec_attr_trig** calls.
- The user-written attribute trigger servers are independent from DCE servers. The trigger servers implement the trigger operations for the attribute types that require attribute trigger processing. These servers are not provided as part of the DCE, but must be written by application developers.

## Defining an Attribute Trigger/Attribute Association

When an attribute is created with the **sec_rgy_attr_update()** call you define the association between the attribute type and an attribute trigger by specifying the following:

- **Trigger Type**—Defines the trigger as a query server (invoked for query operations) or an update server (invoked for updates operations). The trigger type is defined in a **sec_attr_trig_type_t** data type, which is used by a **sec_attr_schema_entry_t** data type.
- **Trigger Binding**—Defines the server binding handle for the attribute trigger. The details of the trigger binding are defined in a number of data types, which are also used by the **sec_attr_schema_entry_t** data type. Trigger binding is described in detail in “Trigger Binding” on page 482.

Only if both pieces of information are provided will the association between the attribute type and the attribute trigger be created. You can associate an attribute trigger to any attribute type of any encoding except for attribute sets.

**Query Triggers:** When you execute a call that accesses an attribute associated with query trigger, the client side attribute lookup code performs the following tasks:

- Binds to the attribute trigger (using a binding from the attribute type's schema entry)
- Makes the remote **sec_attr_trig_query()** call to the attribute trigger server, passing in the attribute keys and optional information provided by the caller
- If the **sec_attr_trig_query()** call is successful, returns the output attribute(s) to the caller

If you execute an **sec_rgy_attr** update call with an attribute type that is associated with a query trigger, not an update trigger, the input attribute values is ignored and a stub attribute instance is created on the named object simply to mark the existence of this attribute on the object. Modifications to the real attribute value must occur at the attribute trigger.

**Update Triggers:** When you execute a call that accesses an attribute associated with an update trigger, the client side attribute update code performs the following tasks:

- Binds to the attribute trigger (using a binding from the attribute type's schema entry)
- Makes the remote **sec_attr_trig_update()** call to the attribute trigger server, passing in the attributes provided by the caller
- If the **sec_attr_trig_update()** call is successful, stores the output attribute(s) in the Registry database and returns the output attribute(s) to the caller.

## Trigger Binding

Two data types are used to define an attribute trigger. The **sec_attr_trig_type_t** defines the type of attribute trigger. The **sec_attr_bind_info_t** data type, illustrated in Figure 102 and described in this section, specifies the attribute trigger's binding.

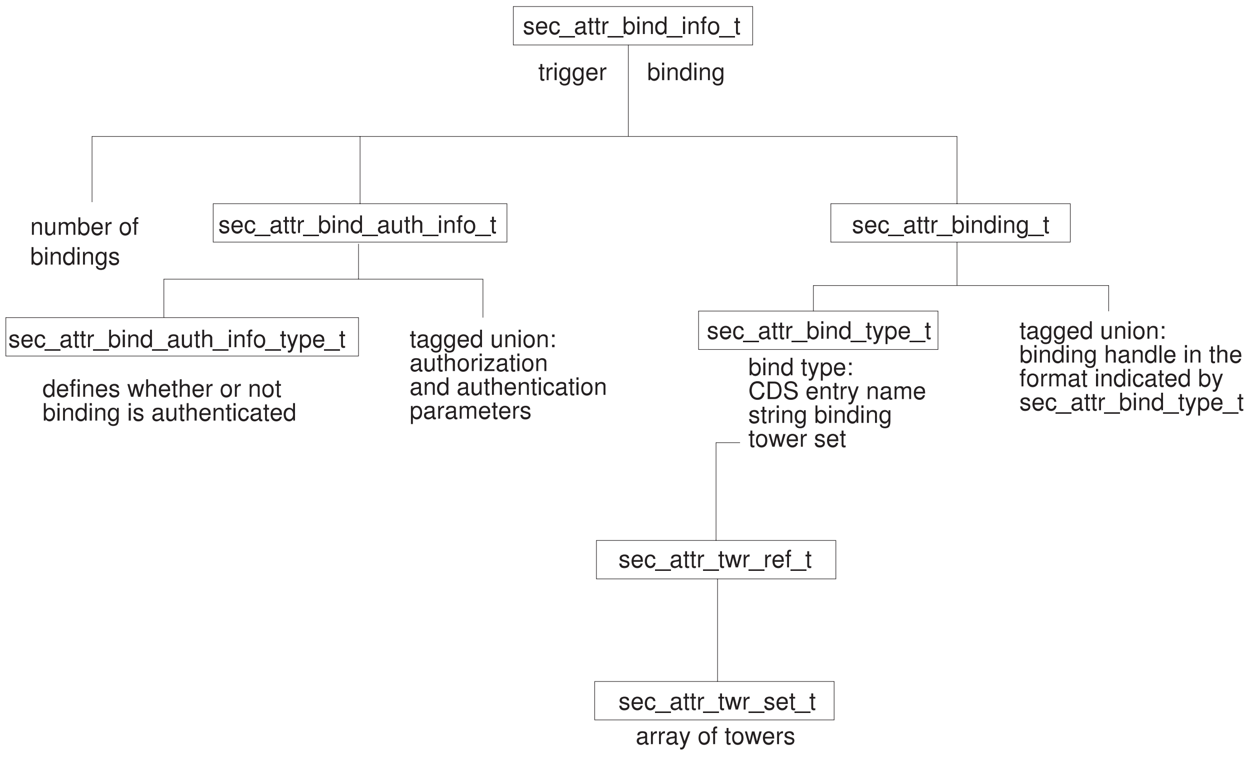

sec_attr_bind_info_t
trigger
binding
number of bindings
sec_attr_bind_auth_info_t
sec_attr_binding_t
sec_attr_bind_auth_info_type_t
defines whether or not binding is authenticated
tagged union: authorization and authentication parameters
sec_attr_bind_type_t
bind type: CDS entry name string binding tower set
tagged union: binding handle in the format indicated by sec_attr_bind_type_t
sec_attr_twr_ref_t
sec_attr_twr_set_t
array of towers

*Figure 102. The sec_attr_bind_info_t Data Type*

The **sec_attr_bind_info_t** data type uses two data types: **sec_attr_binding_t**, which defines the information used to generate binding handle and **sec_attr_bind_auth_info_t**, which defines the binding authentication and authorization information.

**The sec_attr_binding_t Data Type:** To describe the binding handle, the **sec_attr_binding_t** uses a **sec_attr_bind_type_t** data type that specifies the format to the data used to generate the binding handle and a tagged union that contains the binding handle. The binding handle can be generated from any of the following:

- A server directory entry name (used with **rpc_ns_binding_import** calls)

  If the binding information is a server name, call **rpc_ns_binding_import_begin** to establish a context for importing RPC binding handles from the name service database. For the **rpc_ns_binding_import_begin** call, specify the CDS server directory entry name, an entry name syntax value of **rpc_c_ns_syntax_dce**, and **sec_attr_trig** as the interface handle of the interface to import.

- A string binding (used with **rpc_binding_from_string_binding** calls)

  If the binding information is a string binding, call **rpc_binding_from_string_binding** to generate an RPC binding handle.

- An RPC protocol tower set (used with **rpc_tower_to_binding** calls)

  If the binding information is a protocol tower, two additional data types are used to pass in an unallocated array of towers, which the server must then allocate. These data types are: **sec_attr_twr_ref_t** to point to the tower, and **sec_attr_twr_set_t** to define the array of towers.

  Architectural components of the DCE can take advantage of the internal **rpc_tower_to_binding** operation in **rpcpvt.idl** to generate a binding handle from the canonical representation of a protocol tower.

Although the server directory entry name, with the actual server address stored in the Cell Directory Service, is the recommended way to specify an attribute trigger binding handle, prototype applications may want to specify a string binding or protocol tower for convenience.

**The sec_attr_bind_auth_info_t Data Type:** To describe whether or not RPC calls to the server will be authenticated and for authenticated calls, to provide authentication and authorization information, the **sec_attr_bind_auth_info_t** uses the **sec_attr_bind_auth_info_type_t** data type, and a tagged union. The **sec_attr_bind_auth_info_type_t** defines whether or not the call is authenticated. The tagged union contains the authentication and authorization parameters.

Once a binding handle is obtained, call **rpc_binding_set_auth_info** and supply it with the binding handle and authorization and authentication information.

## Access Control on Attributes with Triggers

When a query or update call accesses an attribute associated with an attribute trigger the call checks the ACL of the object with which the attribute is associated to see if the client has the permissions required for the operation. If access is granted, the operation returns binding handle authenticated with the client's login context. This handle is then used to perform the **sec_attr_trig_query** or **sec_attr_trig_update** operation.

Access to information maintained by an attribute trigger is controlled entirely by that attribute trigger. The attribute trigger can choose to implement any authorization mechanism, including none. For example, the attribute trigger can obtain the client's identity from the RPC runtime to perform name-based authentication and perform ACL checks (or any other type of access control mechanism), and it can query the Registry attribute schema for the attribute type's permission set to use for an ACL check. Access control on attribute information stored outside of the Registry database is left to the application designer.

## Calls that Access Attribute Triggers

This section describes the calls that send information to and receive it from attribute triggers.

### Using sec_attr_trig_cursor_t with sec_attr_trig_query()

The **sec_attr_trig_query()** call, which reads attributes associated with a query attribute trigger, uses a cursor of type **sec_attr_trig_cursor_t**. This cursor must be allocated and initialized before it can be used as input to the **sec_attr_trig_query()** call. After use, the resources allocated to the **sec_attr_trig_cursor_t** must be released.

The following calls allocate, initialize, and release a **sec_attr_trig_cursor_t** for use with the **sec_attr_trig_query()** call:

- **sec_attr_trig_cursor_init()**

  The **sec_attr_trig_cursor_init()** call allocates resources to the cursor and initializes the cursor to the first attribute in the list of attributes for the object whose binding handle is specified. This call makes a remote call.

- **sec_attr_trig_cursor_release()**

  The **sec_rgy_attr_cursor_release()** call releases all resources allocated to a **sec_attr_trig_cursor_t** by **sec_attr_trig_cursor_init()**.

### sec_rgy_attr_trig_query() and sec_rgy_attr_trig_update()

The **sec_attr_trig_query()** call reads instances of attributes coded with a trigger type of query for a specified object. It passes an array of **sec_attr_t** values to a query attribute trigger and receives the output parameters back from the server. The **sec_attr_trig_update()** routine passes attributes coded with a trigger type of update to an update attribute trigger for evaluation before the updates are made to the registry.

Both calls are called automatically by the DCE attribute lookup or update code for all schema entries that specify a trigger. Although you should not call them directly, if you are implementing a trigger server, it will receive input from these calls and the attribute trigger's output should be passed back to them. The data received must be in a form accessible to the call and, if it is the result of an update, a form that can be stored in the registry database.

The object whose attribute instances are to be read or updated is identified by:

- The name of the cell in which the object exists.
- The name of the object or a UUID in string format that identifies the object.

### priv_attr_triq_query()

The **priv_attr_trig_query()** call is used by the Privilege service to retrieve trigger attributes and add them to a principal's EPAC. The Privilege service executes this call when it receives a request to add a principal and its extended attribute instances to an EPAC and the attributes are associated with a trigger server. The call passes an array **sec_attr_t** values to the attribute trigger and receives the attribute values back from the trigger server in another array of **sec_attr_t** values. If the principal is being added to a delegation chain, the call also passes the UUIDs of all of the current members of the delegation chain to the trigger server. The trigger server can then evaluate all identities to determine access rights to the requested attributes.

Like the **sec_rgy_attr_trig_query()** and **sec_rgy_attr_trig_update()** calls, you will not call **priv_attr_trig_query()** directly. However, if you are implementing a trigger server, it will receive input from this calls and the attribute trigger's output should be passed back to the call. The data received must be in a form accessible to the call.

## The DCE Attribute API

This DCE attribute calls are not described in detail. This is because with the exception of the calls that bind to a selected database (**dce_attr_sch_bind** and **dce_attr_sch_bind_free**), the **dce_sec_attr** calls are the same as the **sec_rgy_attr_sch** calls. Refer to “Extended Registry Attribute API” on page 465 for information on using each call.

The DCE Attribute API consists of the following calls:

| | |
|---|---|
| **dce_attr_sch_bind** | Returns an opaque handle of type **dce_attr_sch_handle_t** to a schema object specified by name and sets authentication and authorization parameters for the handle. This is the call used to bind to the schema of your choice. |
| **dce_attr_sch_bind_free** | Releases an opaque handle of type **dce_attr_sch_handle_t**. |
| **dce_attr_sch_create_entry** | Creates a schema entry in a schema bound to with **dce_attr_sch_bind**. This call is based on the **sec_rgy_attr_sch_create_entry** and is used in the same way. |
| **dce_attr_sch_update_entry** | Updates a schema entry in a schema bound to with **dce_attr_sch_bind**. This call is based on the **sec_rgy_attr_sch_update_entry** and is used in the same way. |
| **dce_attr_sch_delete_entry** | Deletes a schema entry in a schema bound to with **dce_attr_sch_bind**. This call is based on the **sec_rgy_attr_sch_delete_entry** and is used in the same way. |
| **dce_attr_sch_scan** | Reads a specified number of schema entries. This call is based on the **sec_rgy_attr_sch_scan** and is used in the same way. |
| **dce_attr_sch_cursor_init** | Allocates resources to and initializes a cursor used with **dce_attr_sch_scan**. The **dce_attr_sch_cursor_init** routine makes a remote call that also returns the current number of schema entries in the schema. The **dce_attr_sch_cursor_init** call is based on the **sec_rgy_attr_sch_cursor_init** and is used in the same way. |
| **dce_attr_sch_cursor_alloc** | Allocates resources to a cursor used with **dce_attr_sch_scan**. The **dce_attr_sch_cursor_alloc** routine is a local operation. The **dce_attr_sch_cursor_alloc** call is based on the **sec_rgy_attr_sch_cursor_alloc** and is used in the same way. |
| **dce_attr_sch_cursor_release** | Releases states associated with a cursor created by **dce_attr_sch_cursor_alloc** or **dce_attr_sch_cursor_init**. The **dce_attr_sch_cursor_release** call is based on the **sec_rgy_attr_sch_cursor_release** and is used in the same way. |
| **dce_attr_sch_cursor_reset** | Reinitializes a cursor used with **dce_attr_sch_scan**. The reset cursor can then be reused without releasing and re-allocating. This call is based on the **sec_rgy_attr_sch_cursor_reset** and is used in the same way. |

| | |
|---|---|
| **dce_attr_sch_lookup_by_id** | Reads a schema entry identified by UUID. This call is based on the **sec_rgy_attr_lookup_by_id** and is used in the same way. |
| **dce_attr_sch_lookup_by_name** | Reads a schema entry identified by name. This call is based on the **sec_rgy_attr_sch_lookup_by_name** and is used in the same way. |
| **dce_attr_sch_get_acl_mgrs** | Retrieves the UUIDs of ACL manager types protecting objects dominated by a named schema. This call is based on the **sec_rgy_attr_sch_get_acl_mgrs** and is used in the same way. |
| **dce_attr_sch_aclmgr_strings** | Retrieves the printstrings containing information about ACL manager types protecting objects dominated by a named schema. The printstrings contain the manager's name, help information, and supported permission bits. This call is based on the **sec_rgy_attr_sch_aclmgr_strings** and is used in the same way. |

## Macros to Aid Extended Attribute Programming

The Extended Attribute APIs includes macros to help programmers using the Extended Attribute interfaces. The macros perform a variety of function including:

- Accessing fields in data structures
- Calculating the size of data structures
- Performing semantic and flag checks
- Setting flags

The macros are in **dce/rpcbase.h**, which is derived from **dce/rpcbase.idl**.

The following sections list the definitions of each macro.

## Macros to Access Binding Fields

In the following macro definitions, which are used by a **sec_attr_schema_entry_t** and its equivalent **dce_attr_sch** data type, B is a pointer to a **sec_attr_bind_info_t** data.

```
#define SA_BND_AUTH_INFO(B)  (B)->auth_info
#define SA_BND_AUTH_INFO_TYPE(B) (SA_BND_AUTH_INFO(B)).info_type

#define SA_BND_AUTH_SVR_PNAME_P(B) \
        (SA_BND_AUTH_DCE_INFO(B)).svr_princ_name

#define SA_BND_AUTH_PROT_LEVEL(B) \
        (SA_BND_AUTH_DCE_INFO(B)).protect_level

#define SA_BND_AUTH_AUTHN_SVC(B) \
        (SA_BND_AUTH_DCE_INFO(B)).authn_svc

#define SA_BND_AUTH_AUTHZ_SVC(B) \
        (SA_BND_AUTH_DCE_INFO(B)).authz_svc

#define SA_BND_NUM(B)        (B)->num_bindings
#define SA_BND_ARRAY(B,I)        (B)->bindings[I]
#define SA_BND_TYPE(B,I)        (SA_BND_ARRAY(B,I)).bind_type

#define SA_BND_STRING_P(B,I)  \
        (SA_BND_ARRAY(B,I)).tagged_union.string_binding
```

```
#define SA_BND_SVRNAME_P(B,I)  \
        (SA_BND_ARRAY(B,I)).tagged_union.svrname

#define SA_BND_SVRNAME_SYNTAX(B,I) \
        (SA_BND_SVRNAME_P(B,I))->name_syntax

#define SA_BND_SVRNAME_NAME_P(B,I) \
        (SA_BND_SVRNAME_P(B,I))->name

#define SA_BND_TWRSET_P(B,I)  \
        (SA_BND_ARRAY(B,I)).tagged_union.twr_set

#define SA_BND_TWRSET_COUNT(B,I) (SA_BND_TWRSET_P(B,I))->count
#define SA_BND_TWR_P(B,I,J)  (SA_BND_TWRSET_P(B,I))->towers[J]
#define SA_BND_TWR_LEN(B,I,J)  (SA_BND_TWR_P(B,I,J))->tower_length

#define SA_BND_TWR_OCTETS(B,I,J) \
        (SA_BND_TWR_P(B,I,J))->tower_octet_string
```

## Macros to Access Schema Entry Fields

In the following macro definitions, S is a pointer to a **sec_attr_schema_entry_t** (and its equivalent **dce_attr_sch** data type) and I and J are non-negative integers for array element selection.

```
#define SA_ACL_MGR_SET_P(S)  (S)->acl_mgr_set
#define SA_ACL_MGR_NUM(S)       (SA_ACL_MGR_SET_P(S))->num_acl_mgrs
#define SA_ACL_MGR_INFO_P(S,I)  (SA_ACL_MGR_SET_P(S))->mgr_info[I]
#define SA_ACL_MGR_TYPE(S,I)  (SA_ACL_MGR_INFO_P(S,I))->acl_mgr_type
#define SA_ACL_MGR_PERMS_QUERY(S,I) (SA_ACL_MGR_INFO_P(S,I))->query_permset
#define SA_ACL_MGR_PERMS_UPDATE(S,I) (SA_ACL_MGR_INFO_P(S,I))->update_permset
#define SA_ACL_MGR_PERMS_TEST(S,I) (SA_ACL_MGR_INFO_P(S,I))->test_permset
#define SA_ACL_MGR_PERMS_DELETE(S,I) (SA_ACL_MGR_INFO_P(S,I))->delete_permset
#define SA_TRG_BND_INFO_P(S)  (S)->trig_binding

#define SA_TRG_BND_AUTH_INFO(S)            \
        (SA_BND_AUTH_INFO(SA_TRG_BND_INFO_P(S)))

#define SA_TRG_BND_AUTH_INFO_TYPE(S)       \
        (SA_BND_AUTH_INFO_TYPE(SA_TRG_BND_INFO_P(S)))

#define SA_TRG_BND_AUTH_DCE_INFO(S)        \
        (SA_BND_AUTH_DCE_INFO(SA_TRG_BND_INFO_P(S)))

#define SA_TRG_BND_AUTH_SVR_PNAME_P(S)     \
        (SA_BND_AUTH_SVR_PNAME_P(SA_TRG_BND_INFO_P(S)))

#define SA_TRG_BND_AUTH_PROT_LEVEL(S)      \
        (SA_BND_AUTH_PROT_LEVEL(SA_TRG_BND_INFO_P(S)))

#define SA_TRG_BND_AUTH_AUTHN_SVC(S)       \
        (SA_BND_AUTH_AUTHN_SVC(SA_TRG_BND_INFO_P(S)))

#define SA_TRG_BND_AUTH_AUTHZ_SVC(S)       \
        (SA_BND_AUTH_AUTHZ_SVC(SA_TRG_BND_INFO_P(S)))

#define SA_TRG_BND_NUM(S)                  \
        (SA_BND_NUM(SA_TRG_BND_INFO_P(S)))

#define SA_TRG_BND_ARRAY(S,I)              \
        (SA_BND_ARRAY((SA_TRG_BND_INFO_P(S)),I))

#define SA_TRG_BND_TYPE(S,I)               \
        (SA_BND_TYPE((SA_TRG_BND_INFO_P(S)),I))
```

```
#define SA_TRG_BND_STRING_P(S,I)           \
        (SA_BND_STRING_P((SA_TRG_BND_INFO_P(S)),I))

#define SA_TRG_BND_SVRNAME_P(S,I)          \
        (SA_BND_SVRNAME_P((SA_TRG_BND_INFO_P(S)),I))

#define SA_TRG_BND_SVRNAME_SYNTAX(S,I)     \
        (SA_BND_SVRNAME_SYNTAX((SA_TRG_BND_INFO_P(S)),I))

#define SA_TRG_BND_SVRNAME_NAME_P(S,I)     \
        (SA_BND_SVRNAME_NAME_P((SA_TRG_BND_INFO_P(S)),I))

#define SA_TRG_BND_TWRSET_P(S,I)           \
        (SA_BND_TWRSET_P((SA_TRG_BND_INFO_P(S)),I))

#define SA_TRG_BND_TWRSET_COUNT(S,I)       \
        (SA_BND_TWRSET_COUNT((SA_TRG_BND_INFO_P(S)),I))

#define SA_TRG_BND_TWR_P(S,I,J)            \
        (SA_BND_TWR_P((SA_TRG_BND_INFO_P(S)),I,J))

#define SA_TRG_BND_TWR_LEN(S,I,J)          \
        (SA_BND_TWR_LEN((SA_TRG_BND_INFO_P(S)),I,J))

#define SA_TRG_BND_TWR_OCTETS(S,I,J)       \
        (SA_BND_TWR_OCTETS((SA_TRG_BND_INFO_P(S)),I,J))
```

## Macros to Access Attribute Instance Fields

In the following macro descriptions, S is a pointer to a **sec_attr_t** and I and J are nonnegative integers for array element selection.

```
#define SA_ATTR_ID(S)   (S)->attr_id
#define SA_ATTR_VALUE(S)       (S)->attr_value
#define SA_ATTR_ENCODING(S)  (SA_ATTR_VALUE(S)).attr_encoding

#define SA_ATTR_INTEGER(S)             \
        (SA_ATTR_VALUE(S)).tagged_union.signed_int

#define SA_ATTR_PRINTSTRING_P(S)       \
        (SA_ATTR_VALUE(S)).tagged_union.printstring

#define SA_ATTR_STR_ARRAY_P(S)         \
        (SA_ATTR_VALUE(S)).tagged_union.string_array

#define SA_ATTR_STR_ARRAY_NUM(S) (SA_ATTR_STR_ARRAY_P(S))->num_strings
#define SA_ATTR_STR_ARRAY_ELT_P(S,I) (SA_ATTR_STR_ARRAY_P(S))->strings[I]

#define SA_ATTR_BYTES_P(S)             \
        (SA_ATTR_VALUE(S)).tagged_union.bytes

#define SA_ATTR_BYTES_LEN(S)  (SA_ATTR_BYTES_P(S))->length
#define SA_ATTR_BYTES_DATA(S,I) (SA_ATTR_BYTES_P(S))->data[I]

#define SA_ATTR_IDATA_P(S)             \
        (SA_ATTR_VALUE(S)).tagged_union.idata

#define SA_ATTR_IDATA_CODESET(S) (SA_ATTR_IDATA_P(S))->codeset
#define SA_ATTR_IDATA_LEN(S)  (SA_ATTR_IDATA_P(S))->length
#define SA_ATTR_IDATA_DATA(S,I) (SA_ATTR_IDATA_P(S))->data[I]

#define SA_ATTR_UUID(S)                 \
        (SA_ATTR_VALUE(S)).tagged_union.uuid

#define SA_ATTR_SET_P(S)                \
        (SA_ATTR_VALUE(S)).tagged_union.attr_set

#define SA_ATTR_SET_NUM(S)  (SA_ATTR_SET_P(S))->num_members
#define SA_ATTR_SET_MEMBERS(S,I) (SA_ATTR_SET_P(S))->members[I]

#define SA_ATTR_BND_INFO_P(S)            \
        (SA_ATTR_VALUE(S)).tagged_union.binding

#define SA_ATTR_BND_AUTH_INFO(S)         \
        (SA_BND_AUTH_INFO(SA_ATTR_BND_INFO_P(S)))

#define SA_ATTR_BND_AUTH_INFO_TYPE(S)    \
        (SA_BND_AUTH_INFO_TYPE(SA_ATTR_BND_INFO_P(S)))

#define SA_ATTR_BND_AUTH_DCE_INFO(S)     \
        (SA_BND_AUTH_DCE_INFO(SA_ATTR_BND_INFO_P(S)))

#define SA_ATTR_BND_AUTH_SVR_PNAME_P(S)  \
        (SA_BND_AUTH_SVR_PNAME_P(SA_ATTR_BND_INFO_P(S)))

#define SA_ATTR_BND_AUTH_PROT_LEVEL(S)   \
        (SA_BND_AUTH_PROT_LEVEL(SA_ATTR_BND_INFO_P(S)))

#define SA_ATTR_BND_AUTH_AUTHN_SVC(S)    \
        (SA_BND_AUTH_AUTHN_SVC(SA_ATTR_BND_INFO_P(S)))

#define SA_ATTR_BND_AUTH_AUTHZ_SVC(S)    \
        (SA_BND_AUTH_AUTHZ_SVC(SA_ATTR_BND_INFO_P(S)))

#define SA_ATTR_BND_NUM(S)               \
        (SA_BND_NUM(SA_ATTR_BND_INFO_P(S)))

#define SA_ATTR_BND_ARRAY(S,I)           \
        (SA_BND_ARRAY((SA_ATTR_BND_INFO_P(S)),I))

#define SA_ATTR_BND_TYPE(S,I)            \
        (SA_BND_TYPE((SA_ATTR_BND_INFO_P(S)),I))

#define SA_ATTR_BND_STRING_P(S,I)        \
        (SA_BND_STRING_P((SA_ATTR_BND_INFO_P(S)),I))

#define SA_ATTR_BND_SVRNAME_P(S,I)       \
        (SA_BND_SVRNAME_P((SA_ATTR_BND_INFO_P(S)),I))

#define SA_ATTR_BND_SVRNAME_SYNTAX(S,I)  \
        (SA_BND_SVRNAME_SYNTAX((SA_ATTR_BND_INFO_P(S)),I))

#define SA_ATTR_BND_SVRNAME_NAME_P(S,I)  \
        (SA_BND_SVRNAME_NAME_P((SA_ATTR_BND_INFO_P(S)),I))

#define SA_ATTR_BND_TWRSET_P(S,I)        \
        (SA_BND_TWRSET_P((SA_ATTR_BND_INFO_P(S)),I))

#define SA_ATTR_BND_TWRSET_COUNT(S,I)    \
        (SA_BND_TWRSET_COUNT((SA_ATTR_BND_INFO_P(S)),I))

#define SA_ATTR_BND_TWR_P(S,I,J)         \
        (SA_BND_TWR_P((SA_ATTR_BND_INFO_P(S)),I,J))

#define SA_ATTR_BND_TWR_LEN(S,I,J)       \
        (SA_BND_TWR_LEN((SA_ATTR_BND_INFO_P(S)),I,J))
```

```
#define SA_ATTR_BND_TWR_OCTETS(S,I,J)     \
        (SA_BND_TWR_OCTETS((SA_ATTR_BND_INFO_P(S)),I,J))
```

## Binding Data Structure Size Calculation Macros

The following macros are supplied to calculate the size of data types that hold binding information. The macros work with the Extended Registry Attribute API data types and their equivalent **dce_attr_sch** data types.

```
/*
 * SA_BND_INFO_SIZE(N) - calculate the size required
 * for a sec_attr_bind_info_t with N bindings.
 */
#define SA_BND_INFO_SIZE(N) ( sizeof(sec_attr_bind_info_t) + \
        (((N) - 1) * sizeof(sec_attr_binding_t)) )

/*
 * SA_TWR_SET_SIZE(N) - calculate the size required
 * for a sec_attr_twr_set_t with N towers.
 */
#define SA_TWR_SET_SIZE(N) ( sizeof(sec_attr_twr_set_t) +     \
        (((N) - 1) * sizeof(sec_attr_twr_ref_t)) )

/*
 * SA_TWR_SIZE(N) - calculate the size required
 * for a twr_t with a tower_octet_string of length N.
 */
#define SA_TWR_SIZE(N) ( sizeof(twr_t) + (N) - 1 )
```

## Schema Entry Data Structure Size Calculation Macros

The following macro is supplied to calculate the size of an **sec_attr_acl_mgr_info_set_t**.

```
/*
 * SA_ACL_MGR_SET_SIZE(N) - calculate the size required
 * for a sec_attr_acl_mgr_info_set_t with N acl_mgrs.
 */
#define SA_ACL_MGR_SET_SIZE(N) ( sizeof(sec_attr_acl_mgr_info_set_t) + \
        (((N) - 1) * sizeof(sec_attr_acl_mgr_info_p_t)) )
```

## Attribute Instance Data Structure Size Calculation Macros

The following macros are supplied to calculate the size of data types that hold attribute information.

```
/*
 * SA_ATTR_STR_ARRAY_SIZE(N) - calculate the size required
 * for a sec_attr_enc_str_array_t with N sec_attr_enc_printstring_p_t-s.
 */
#define SA_ATTR_STR_ARRAY_SIZE(N) ( sizeof(sec_attr_enc_str_array_t) + \
        (((N) - 1) * sizeof(sec_attr_enc_printstring_p_t)) )

/*
 * SA_ATTR_BYTES_SIZE(N) - calculate the size required
 * for a sec_attr_enc_bytes_t with byte string length of N.
 */
#define SA_ATTR_BYTES_SIZE(N) ( sizeof(sec_attr_enc_bytes_t) + (N) - 1 )

/*
 * SA_ATTR_IDATA_SIZE(N) - calculate the size required
 * for a sec_attr_i18n_data_t with byte string length of N.
 */
#define SA_ATTR_IDATA_SIZE(N) ( sizeof(sec_attr_i18n_data_t) + (N) - 1 )

/*
 * SA_ATTR_SET_SIZE(N) - calculate the size required
 * for a sec_attr_enc_attr_set_t with N members (uuids).
 */
#define SA_ATTR_SET_SIZE(N) ( sizeof(sec_attr_enc_attr_set_t) + \
        (((N) - 1) * sizeof(uuid_t)) )
```

## Binding Semantic Check Macros

The following macros are supplied to check the semantics of entries in the binding fields. The macros work with the Extended Registry Attribute API data types and their equivalent **dce_attr_sch** data types.

```
/*
 * SA_BND_AUTH_INFO_TYPE_VALID(B) - evaluates to TRUE (1)
 * if the binding auth_info type is valid; FALSE (0) otherwise.
 * B is a pointer to a sec_attr_bind_info_t.
 */
#define SA_BND_AUTH_INFO_TYPE_VALID(B) ( \
    (SA_BND_AUTH_INFO_TYPE(B)) == sec_attr_bind_auth_none || \
    (SA_BND_AUTH_INFO_TYPE(B)) == sec_attr_bind_auth_dce ? true : false )

/*
 * SA_BND_AUTH_PROT_LEV_VALID(B) - evaluates to TRUE (1)
 * if the binding auth_info protect_level is valid; FALSE (0) otherwise.
 * B is a pointer to a sec_attr_bind_info_t.
 */
#define SA_BND_AUTH_PROT_LEV_VALID(B) ( \
    (SA_BND_AUTH_PROT_LEVEL(B)) == rpc_c_protect_level_default  || \
    (SA_BND_AUTH_PROT_LEVEL(B)) == rpc_c_protect_level_none  || \
    (SA_BND_AUTH_PROT_LEVEL(B)) == rpc_c_protect_level_connect  || \
    (SA_BND_AUTH_PROT_LEVEL(B)) == rpc_c_protect_level_call  || \
    (SA_BND_AUTH_PROT_LEVEL(B)) == rpc_c_protect_level_pkt  || \
    (SA_BND_AUTH_PROT_LEVEL(B)) == rpc_c_protect_level_pkt_integ || \
    (SA_BND_AUTH_PROT_LEVEL(B)) == rpc_c_protect_level_pkt_privacy ? \
    true : false )

/*
 * SA_BND_AUTH_AUTHN_SVC_VALID(B) - evaluates to TRUE (1)
 * if the binding auth_info authentication service is valid;
 * FALSE (0) otherwise.
 * B is a pointer to a sec_attr_bind_info_t.
 */
#define SA_BND_AUTH_AUTHN_SVC_VALID(B) ( \
    (SA_BND_AUTH_AUTHN_SVC(B)) == rpc_c_authn_none  || \
    (SA_BND_AUTH_AUTHN_SVC(B)) == rpc_c_authn_dce_secret || \
    (SA_BND_AUTH_AUTHN_SVC(B)) == rpc_c_authn_dce_public || \
    (SA_BND_AUTH_AUTHN_SVC(B)) == rpc_c_authn_dce_dummy || \
    (SA_BND_AUTH_AUTHN_SVC(B)) == rpc_c_authn_dssa_public || \
    (SA_BND_AUTH_AUTHN_SVC(B)) == rpc_c_authn_default ? \
    true : false )

/*
 * SA_BND_AUTH_AUTHZ_SVC_VALID(B) - evaluates to TRUE (1)
 * if the binding auth_info authorization service is valid;
 * FALSE (0) otherwise.
 * B is a pointer to a sec_attr_bind_info_t.
 */
#define SA_BND_AUTH_AUTHZ_SVC_VALID(B) ( \
    (SA_BND_AUTH_AUTHZ_SVC(B)) == rpc_c_authz_none || \
    (SA_BND_AUTH_AUTHZ_SVC(B)) == rpc_c_authz_name || \
    (SA_BND_AUTH_AUTHZ_SVC(B)) == rpc_c_authz_dce ? \
    true : false )
```

## Schema Entry Semantic Check Macros

The following macros are supplied to check the semantics of schema entry fields. In the macros, S is a pointer to a **sec_attr_schema_entry_t** and its equivalent **dce_attr_sch** data type.

```
#define SA_TRG_BND_AUTH_INFO_TYPE_VALID(S) \
        (SA_BND_AUTH_INFO_TYPE_VALID(SA_TRG_BND_INFO_P(S)))

#define SA_TRG_BND_AUTH_PROT_LEV_VALID(S)      \
        (SA_BND_AUTH_PROT_LEV_VALID(SA_TRG_BND_INFO_P(S)))

#define SA_TRG_BND_AUTH_AUTHN_SVC_VALID(S) \
        (SA_BND_AUTH_AUTHN_SVC_VALID(SA_TRG_BND_INFO_P(S)))

#define SA_TRG_BND_AUTH_AUTHZ_SVC_VALID(S) \
        (SA_BND_AUTH_AUTHZ_SVC_VALID(SA_TRG_BND_INFO_P(S))
```

## Attribute Instance Semantic Check Macros

The following macros are supplied to check the semantics of entries in the attribute instance fields. In the following macros, S is a pointer to a **sec_attr_t**. F is a **sec_attr_trigs_types_flags_t**.

```
#define SA_ATTR_BND_AUTH_INFO_TYPE_VALID(S) \
        (SA_BND_AUTH_INFO_TYPE_VALID(SA_ATTR_BND_INFO_P(S)))

#define SA_ATTR_BND_AUTH_PROT_LEV_VALID(S) \
        (SA_BND_AUTH_PROT_LEV_VALID(SA_ATTR_BND_INFO_P(S)))

#define SA_ATTR_BND_AUTH_AUTHN_SVC_VALID(S) \
        (SA_BND_AUTH_AUTHN_SVC_VALID(SA_ATTR_BND_INFO_P(S)))

#define SA_ATTR_BND_AUTH_AUTHZ_SVC_VALID(S) \
        (SA_BND_AUTH_AUTHZ_SVC_VALID(SA_ATTR_BND_INFO_P(S))

#define SA_SCH_FLAG_IS_SET(S,F)            \
        (((S)->schema_entry_flags & (F)) == (F))

#define SA_SCH_FLAG_IS_SET_UNIQUE(S)            \
        (SA_SCH_FLAG_IS_SET((S),sec_attr_sch_entry_unique))

#define SA_SCH_FLAG_IS_SET_MULTI_INST(S)      \
        (SA_SCH_FLAG_IS_SET((S),sec_attr_sch_entry_multi_inst))

#define SA_SCH_FLAG_IS_SET_RESERVED(S)      \
        (SA_SCH_FLAG_IS_SET((S),sec_attr_sch_entry_reserved))

#define SA_SCH_FLAG_IS_SET_USE_DEFAULTS(S) \
        (SA_SCH_FLAG_IS_SET((S),sec_attr_sch_entry_use_defaults))
```

## Schema Entry Flag Set and Unset Macros

The following macros set and unset flag(s) in the schema entry **schema_entry_flags** field. In the following macros, S is a pointer to a **sec_attr_schema_entry_t**.

```
/*
 * Macro's to set the flags.
 */
#define SA_SCH_FLAG_SET(S, FLAG) ((S)->schema_entry_flags |= (FLAG))

#define SA_SCH_FLAG_SET_UNIQUE(S)      \
        (SA_SCH_FLAG_SET((S),sec_attr_sch_entry_unique))

#define SA_SCH_FLAG_SET_MULTI_INST(S)        \
        (SA_SCH_FLAG_SET((S),sec_attr_sch_entry_multi_inst))

#define SA_SCH_FLAG_SET_RESERVED(S)    \
        (SA_SCH_FLAG_SET((S),sec_attr_sch_entry_reserved))

#define SA_SCH_FLAG_SET_USE_DEFAULTS(S)      \
        (SA_SCH_FLAG_SET((S),sec_attr_sch_entry_use_defaults))

/*
 * Macro's to unset the flags.
 */
#define SA_SCH_FLAG_UNSET(S, FLAG) ((S)->schema_entry_flags &=  (FLAG))

#define SA_SCH_FLAG_UNSET_UNIQUE(S)      \
        (SA_SCH_FLAG_UNSET((S),sec_attr_sch_entry_unique))

#define SA_SCH_FLAG_UNSET_MULTI_INST(S)  \
        (SA_SCH_FLAG_UNSET((S),sec_attr_sch_entry_multi_inst))

#define SA_SCH_FLAG_UNSET_RESERVED(S)  \
        (SA_SCH_FLAG_UNSET((S),sec_attr_sch_entry_reserved))

#define SA_SCH_FLAG_UNSET_USE_DEFAULTS(S)      \
        (SA_SCH_FLAG_UNSET((S),sec_attr_sch_entry_use_defaults))
```

## Schema Trigger Entry Flag Check Macros

The following macros evaluate to TRUE if the requested flag(s) is set in the schema entry **trig_types** field. In the following macros, S is a pointer to a **sec_attr_schema_entry_t** and F is a **sec_attr_trigs_types_flags_t**.

```
#define SA_SCH_TRIG_FLAG_IS_SET(S,F)  \
        (((S)->trig_types & (F)) == (F))

#define SA_SCH_TRIG_FLAG_IS_NONE(S)  \
        (SA_SCH_TRIG_FLAG_IS_SET((S),sec_attr_trig_type_none))

#define SA_SCH_TRIG_FLAG_IS_QUERY(S)  \
        (SA_SCH_TRIG_FLAG_IS_SET((S),sec_attr_trig_type_query))

#define SA_SCH_TRIG_FLAG_IS_UPDATE(S) \
        (SA_SCH_FLAG_IS_SET((S),sec_attr_trig_type_update))
```

## Utilities to Use with Extended Attribute Calls

The Extended Attribute APIs includes utilities to help programmers using the Extended Attribute interfaces. These utilities are:

| | |
|---|---|
| **sec_attr_util_alloc_copy** | Copies data from one **sec_attr_t** data to another. |
| **sec_attr_util_free** | Frees memory allocated to a **sec_attr_t** by the **sec_attr_util_alloc_copy()** function. |
| **sec_attr_util_inst_free_ptrs** | Frees non-null pointers in a **sec_attr_t**. |
| **sec_attr_util_inst_free** | Frees non-null pointers in a **sec_attr_t** and the pointer to the **sec_attr_t** itself. |

|  |  |
|---|---|
| **sec_attr_util_sch_ent_free_ptrs** | Frees non-null pointers in a **sec_attr_schema_entry_t**. |
| **sec_attr_util_sch_ent_free** | Frees non-null pointers in a **sec_attr_schema_entry_t** and the pointer to the **sec_attr_schema_entry_t** itself. The utility also works with the equivalent **dce_attr_sch** data type. |

# Chapter 30. The Login Application Program Interface

This chapter shows you how to use the DCE Login API and related APIs so that you can have your DCE application login to DCE Security and establish its login context. It includes information on:

- Establishing login contexts
- Handling expired certificates of identity
- Importing and exporting contexts
- Changing a groupset
- Automatically logging into DCE
- Miscellaneous login API functions.

Read this chapter if you are interested in having your application login on its own or any of the other tasks related to DCE login.

The Login API communicates with the Security Server to establish, and possibly change, a principal's login context. A login context contains the information necessary for a principal to qualify for, although not necessarily be granted access to, network services and possibly local resources. Login context information normally includes the following:

- Identity information concerning the principal, including its certificate of identity (in Shared-Secret Authentication, this is the TGT), its EPAC, and Registry policy information, such as the maximum lifetime of certificates of identity.
- The context state (that is, whether the Authentication Service has validated the context or not).
- The source of authentication information, that is, the network Authentication Service.

## Establishing Login Contexts

The basic procedure by which a network login context is established (described in detail in Chapter 24, "Authentication" on page 413) is as follows:

1. The client calls **sec_login_setup_identity()** specifying the name of the principal whose network identity is to be established. Memory is allocated to receive the principal's login context.
2. The client calls **sec_login_validate_identity()**, which:
    a. Forwards a TGT request encrypted with the user's secret key and with a random key, to the Authentication Service, which decrypts the request, authenticates the principal, and returns a Ticket-Granting Ticket for the principal.
    b. The client's security runtime then decrypts the TGT and forwards it to the Privilege Service, which creates an EPAC for the principal and encloses it in a Privilege-Ticket Granting Ticket (PTGT), which is returned to the client's security runtime.
    c. The runtime decrypts the message containing the PTGT and returns information about the source of the authentication information to the API (if the authentication information comes from the network Security Server, then the login context is validated).
3. Finally, the client calls **sec_login_set_context()**, which establishes a validated login context that other processes may refer and use.

In "A Walkthrough of User Authentication" on page 416, it was mentioned that one function of **sec_login_valid_and_cert_ident()** is demonstrating that a valid trust path exists between the Authentication Service and the host computer where the principal logs in. After setting up and validating a login context, any application that sets identity information (that is, establishes a login context) for local processes should check to ensure that the server providing the certificate of identity is sound. This demonstrates that the trust path between the client and the Authentication Service is valid.

## Validating the Login Context and Certifying the Security Server

A validated login context is one that is regarded as sound by the local Security runtime. A validated and certified login context is one that is sound, and was likely issued by a valid Security Server. Certifying that the Security Server is valid prevents faked identity information from being distributed to local processes.

For example, a spurious server may collaborate with a dishonest user in order to obtain an identity with comprehensive permissions (for example, an authorized user identity). With such an identity, the dishonest user could gain access to sensitive local objects, such as key-storage files for server principals that run on the host. (Servers running on other hosts would not trust this principal, however, because it does not know their keys.) Of course, if a spurious server can return to the application a ticket encrypted with the hosts's secret key, it means the server has access to the host's key — but in that case, network security has already been seriously undermined.

When an application needs to certify the originator of a certificate of identity, it may call **sec_login_certify_identity()**. This routine makes an authenticated RPC to the local **dced** daemon to acquire a ticket to the host principal. If **dced** succeeds in decrypting the message containing the ticket, the server that granted the certificate of identity must know the host principal's secret key. This evidence indicates that it is a valid Security Server. Because **dced** runs with the required authority to access the host's key, the process calling **sec_login_certify_identity()** need not.

The **sec_login_valid_and_cert_ident()** is similar to **sec_login_certify_identity()**, except that it combines the validation and certification procedures (and therefore, the password of the principal that is logging in must be known to the process making this call). The **sec_login_valid_and_cert_ident()** routine calls the Security Server for a ticket to the host and attempts decryption. The process calling **sec_login_valid_and_cert_ident()** must have access to the host's secret key, and so must possess the proper authority.

**Note:** Because system login programs should not set local identities derived from an uncertified context, all Login API routines that return data from an uncertified context set a status code indicating that the login context is not certified.

## Validating the Login Context Without Certifying the Security Server

An application that does not use login contexts to set local identity information does not need to certify its login contexts. Since an illegitimate Security Server is unlikely to know the key of a remote server principal with which the application might communicate, the application will simply be refused the service requested from the remote server principal. If local operating system identity information is assumed to be neither of interest nor of concern to an application, it may call **sec_login_validate_identity()**, which does not attempt to verify the Security Server's knowledge of the host principal's key.

Unlike **sec_login_certify_identity()** and **sec_login_valid_and_cert_ident()**, the **sec_login_validate_identity()** routine does not acquire a PTGT. Instead, the PTGT is acquired when the application first makes an authenticated RPC.

**Note:** The **sec_login_validate_identity()** and **sec_login_valid_and_cert_ident()** APIs will destroy the input password. If an application needs to refer to the password, a backup copy of the password should be maintained.

## Example of a System Login Program

Figure 103 contains an example of a system login program that obtains a login context that can be trusted for both network and local operations. It must be issued from a privileged user ID.

**Note:** A function call appearing in the following example, **sec_login_purge_context()**, is described in "Miscellaneous Login API Functions" on page 499.

```
if (sec_login_setup_identity(principal,sec_login_no_flags,
    &login_context,&st))
{
   ...get password...

   if (sec_login_valid_and_cert_ident(login_context, password,
       &reset_passwd, &auth_src, &st))
   {
      if(auth_src==sec_login_auth_src_network)
      {
         if (GOOD_STATUS(&st)
            sec_login_set_context(login_context);
      }
   }
   if (reset_passwd)
   {
      ...reset the user's password...

      if (passwd_reset_fails)
      {
         sec_login_purge_context(login_context)

         ...application login-failure actions...
      }

      ...application-specific login-valid actions...
   }
}
```

*Figure 103. Example of a System Login Program from a Privileged User ID*

## Establishing the Initial Context

An application calls **sec_login_setup_identity()** so that it can make other authenticated RPC calls. However, **sec_login_setup_identity()** is itself a local interface to an authenticated RPC, and authenticated RPC needs a validated login context to run. For applications like system login, the daemon **dced** supplies the validated context. However, **dced** must be running before any other daemons can be started.

## Handling Expired Certificates of Identity

For a principal to use an intercepted certificate of identity, it must successfully decrypt it. To make the task of decryption more difficult, a certificate of identity has a limited life span. Once it expires, the associated login context is no longer valid.

Because this security feature may inconvenience users, an application may warn a user when the certificate of identity is about to expire. The **sec_login_get_expiration()** routine returns the expiration date of a certificate of identity. When a certificate of identity is about to expire, the application may call **sec_login_refresh_identity()**, to refresh any login context.

Similarly, a server principal may need to determine whether a certificate of identity could expire during a long network operation. If so, it needs to refresh it to ensure that the operation is not prevented from completing. Following is an example:

```
sec_login_get_expiration (login_context,&expire_time,&st);

if (expire_time < (current_time + operation_duration))
{
   if (sec_login_refresh_identity(login_context,&st))
   {
      ...identity has changed and must be validated again...
   }
   else
   {
      ...login context cannot be renewed...

      exit(0);
   }
}

operation();
```

*Figure 104. An Example of Handling Expired Certificates of Identity*

Note that only long running application servers whose up time exceeds the expiration time for a login context (default is 8 hours) needs to handle expired certificates as in the above example.

Because **sec_login_refresh_identity()** acquires a certificate of identity, refreshed contexts must be revalidated with **sec_login_validate_identity()** or **sec_login_valid_and_cert()** before they can be used.

## Importing and Exporting Contexts

Under some circumstances, an application may need two processes to run using the same login context. A process may acquire its login context in a form suitable for imparting to another process by calling **sec_login_export_context()**. This call collects the login context from the local context cache and loads it into a buffer. Another process can then call **sec_login_import_context()** to unpack the buffer and create its own login context cache to store the imported context. Because the context has already been validated, the process that imports it can use it immediately. (The CDS Clerk is an example of a context importer.)

These operations are strictly local; the exporting and importing processes must be running on the same host. In addition, a process cannot export a private context. A *private context* is created when you use **sec_login_setup_identity()** with the **sec_login_credentials_private** bit (in the *flags* parameter) set to on. When this happens, the login context that is created and validated cannot be *retained*; that is, it cannot be passed from one address space to another as a login context marked private cannot become the network context. That is, the **sec_login_set_context()** call which sets the KRB5CCNAME environment variable cannot be issued, and thus the login context cannot be passed between different address spaces.

Note that importing the login context only enables you to obtain the name of the credential cache file. If you try to use the credential cache file that is pointed to by the login context, you require read and write permissions to that credential cache file. Thus, importing a login context does not imply that you can just pick a login context and use another principal's login credential cache file.

## Changing a Groupset

The **sec_login_newgroups()** routine enables a principal to assume the minimum groupset that is required to accomplish a given task. For example, a user may have privilege attributes that include membership in an administrative group associated with a comprehensive permission set and membership in a user group associated with a more restricted permission set. Such a user may not want the permissions associated with the administrative group, except when those permissions are essential to an administrative task. This avoids inadvertently damaging objects that are accessible to members of the administrative group, but are not accessible to members of the user group.

To offer users the capability of removing groups from their groupsets, an application could use the Login API as shown in the following example.

**Note:** Two of the function calls that appear in the following example, **sec_login_get_current_context()** and **sec_login_inquire_net_info()**, are described in "Miscellaneous Login API Functions."

```
sec_login_get_current_context(&login_context,&st);

sec_login_inquire_net_info(login_context,&net_info,&st);

for (i=0; i < num_groups; i++)
{
   ...query whether the user wants to discard any of the current
   group memberships. Copy new group set to the new_groups array...
}

if (!sec_login_newgroups(login_context,sec_login_no_flags,
    num_new_groups, new_groups, &restricted_context,&st))
{
   if (st == sec_login_s_groupset_invalid)

      printf("Newgroupsetinvalid\n");

   ...application-specific error handling...
}
```

*Figure 105. Removing Groups from Their Groupsets*

Note that the **sec_login_newgroups()** call can only return a restricted groupset: it cannot return a groupset larger than the one associated with the login context that is passed to it. This routine also enables the calling process to flag the new login context as private to the calling process.

## Miscellaneous Login API Functions

This section describes a few miscellaneous Login API routines, some of which have appeared previously in examples in this chapter.

## Getting the Current Context

The **sec_login_get_current_context()** routine returns a handle to the login context for the currently established principal. This routine is useful for several Login API functions that take a login context handle as input.

## Getting Information from a Login Context

The **sec_login_inquire_net_info()** routine returns a data structure comprising the principal's EPAC, account expiration date, password expiration date, and identity expiration date. The **sec_login_free_net_info()** frees the memory allocated to this data structure.

Note that once **sec_login_set_context()** has been used to set the network credentials to a particular login context, **sec_login_get_current_context()** returns a handle to that particular login context, even though the login context may no longer be valid. Therefore, you should verify whether the login context returned by **sec_login_get_current_context()** is still valid by issuing the **sec_login_get_expiration()** call.

## Getting Group Information for Local Process Identities

The **sec_login_get_groups()** call returns password or group information from the network Registry, if that service is available.

## Releasing and Deleting a Context

When a process is finished using a login context, it can call **sec_login_release_context()** to free storage occupied by the context handle. When a process releases a login context, the context is still available to other processes that use it. If an application needs to destroy a login context, it can call **sec_login_purge_context()**, which also frees storage occupied by the handle. Because a destroyed context is unavailable to all processes that use it, application developers should be careful when using **sec_login_purge_context()** to destroy the current context.

# Chapter 31. The Key Management Application Program Interface

Every principal has an entry in the Registry database that specifies a secret key. In the case of an interactive principal (that is, a user), the secret key is derived from the principal's password. Just as users need to keep their passwords secure by memorizing them (rather than writing them down, for example), a noninteractive principal also needs to be able to store and retrieve its secret key in a secure manner. The Key Management API provides simple key management functions for noninteractive principals.

While the key management routines themselves are relatively secure, it is the application's responsibility to ensure the security of the file or other device used to store the key. By default, server principals that run on the same computer share a local key file; however, the Key Management API also allows principals to specify an alternative local file.

When users change their passwords, they are free to forget their old passwords. When a noninteractive principal changes its secret key, however, there may be clients with valid tickets to that principal that are encrypted with the old key. To save clients the trouble of requesting new tickets to a noninteractive principal when the principal's key has changed, every key is flagged with a version number. Old key versions are kept until all tickets that could have been encrypted with that key have expired.

Finally, if a noninteractive principal's key has been compromised, it can be invalidated (along with all the corresponding tickets held by any clients) by simply deleting it from the local key storage.

**Notes:**

1. The Key Management API is for use only by applications using the DCE Shared-Secret Authentication protocol and the key-type DES (Data Encryption Standard).
2. Your server applications should not have more than one thread accessing a given keytab file. If multiple key management threads are necessary, then use multiple keytab files, one keytab file per thread.

## Retrieving a Key

The key management API provides two functions for retrieving a key from the local key storage. The **sec_key_mgmt_get_key()** function returns a specified key version for a specified principal. The meaning of specifying version 0 (zero) in this routine varies depending on the authentication protocol in effect. (If the protocol is DCE Shared-Secret, the value 0 for the version identifier means the version that was most recently added to the local storage.) In any case, a principal's login is almost always successful if the principal uses the version 0 key.

When there are valid tickets that are encrypted with different key versions, an application may need to retrieve more than one key version. In that case, the application can call **sec_key_mgmt_initialize_cursor()** to set a cursor in the local storage to the first suitable entry corresponding to the named principal and key type, and then call **sec_key_mgmt_get_next_key()** to get all versions of that key in storage. The application can then call **sec_key_mgmt_release_cursor()**, which disposes of information associated with the cursor. Neither of the key-retrieval routines can return keys that have been explicitly deleted, or that have been *garbage collected* after expiring.

The two key-retrieval functions dynamically allocate the memory for the returned keys. To enable the efficient allocation of memory, an application can call **sec_key_mgmt_free_key()**, which frees the memory occupied by the key and returns it to the allocation pool.

## Changing a Key

The **sec_key_mgmt_change_key()** function communicates with the Registry to change the principal's key to a specified string, and also places the new string in the local key storage. *Local key storage* is a flat file on the host machine, as opposed to the machine where the Security daemon resides, that is used for storing passwords, or keys, for DCE principals that cannot enter a password interactively. This refers to most server applications that have to *log in* and do not want to supply their passwords as startup parameters. The *keydata* input argument for this call can be a new key that the application specifies, or a random key returned by the **sec_key_mgmt_gen_rand_key()** routine. An application can call **sec_key_mgmt_get_next_kvno()** to determine the next key version number that should be assigned to the new key, so that it can refer to this key version when retrieving a key.

In some circumstances, a principal may need to change its key in the local key storage, but not immediately update the Registry database. For example, a database application can maintain replicas of a master database that are managed by servers running on different computers. If these servers all provide exactly the same service, it makes sense for them to share the same key (meaning that they share the same principal identity). This way, a user with a ticket to the principal can be directed to whichever server is least busy.

When the Registry database obtains a new key for a principal, the Authentication Service can immediately begin issuing tickets to the principal that are encrypted under the new key. However, suppose the master for a single-principal replicated service were to call **sec_key_mgmt_change_key()** and a client presented a ticket encoded with the latest key to a replica that had not yet learned that key. The replica would refuse service, even though the ticket was valid. If an application employs replicated servers that are also instances of a single principal identity, the application should:

1. Generate a new key by calling **sec_key_mgmt_gen_rand_key()**. This routine simply returns a key to the calling process, without updating the Registry or local storage.
2. Disseminate the new key to all replicas.
3. Cause the replicas to call **sec_key_mgmt_set_key()**. This call updates the local storage to the new key, but does not update the Registry database entry for the principal. The key version specified in this routine must not be 0. The replicas should notify the master when they have completed setting their local stores to the new key.
4. Cause the master to call **sec_key_mgmt_change_key()** after all replicas have set the new key locally, thereby updating both the master's local storage and the Registry database entry. Here again, the key version must not be 0.

If the master and each replica has its own principal identity, each server can call **sec_key_mgmt_change_key()** without coordinating this activity with any others.

## Automatic Key Management

It is sometimes convenient for a principal to be able to change its key on a schedule determined by the password expiration policy for that principal, rather than to rely on a network administrator to decide when this should be done. In this case, the application can call **sec_key_mgmt_manage_key()**. This function calls **sec_key_mgmt_gen_rand_key()** shortly before the current key is due to expire, updates both the local key storage and the Registry database entry with the new key, and then calls **sec_key_mgmt_garbage_collect()** to discard any obsolete keys. This function runs indefinitely; it will never return during normal operation and so should be called from a thread dedicated to key management. It is not intended for use by server principals that share the same key.

## Deleting Expired Keys

In order to prevent service interruptions, the Key Management API does not immediately discard keys that have been replaced; instead, it maintains the keys, with a version number and key-type identifier, in the local key storage. However, after a key has been out of use for longer than the maximum life of a ticket to the principal, it is no longer possible that any client of that principal has a valid ticket encrypted with that key. At this time, the principal may have its key storage *garbage* collected. The **sec_key_mgmt_garbage_collect()** routine collects garbage in the local key storage by deleting all keys older than the maximum ticket lifetime for the cell. The *garbage_collect_time* argument, which is returned by **sec_key_mgmt_change_key()**, specifies when key-storage garbage will be collected.

## Deleting a Compromised Key

When a principal's key has been compromised, it should be deleted as soon as the damage has been discovered, in order to prevent another party from masquerading as that principal. Two routines delete a principal's key:

- The **sec_key_mgmt_delete_key()** routine removes all key types having the specified key version identifier from the local key storage, thus invalidating all existing tickets encrypted with that key.
- The **sec_key_mgmt_delete_key_type()** routine removes only a specified version of a specified key type.

If the compromised key is the current one, the application should first change the key with **sec_key_mgmt_change_key()**. It is not an error for a process to delete the current key as long as it is done after the login context has been established, but it may inconvenience valid clients of a service. The inconvenience may be justified, however, if the application data is sensitive.

Since an application may have no means to discover that its key has been compromised, the **rgy_edit** and **dcecp** tools provide interfaces that call **sec_key_mgmt_delete_key()**, **sec_key_mgmt_change_key()**, and **sec_key_mgmt_gen_rand_key()** so that a network administrator, who is more likely to detect that a key has been compromised, may handle a security breach of this kind. As an alternative, the application may provide user interfaces to these routines.

# Chapter 32. The Access Control List Application Program Interfaces

Normally, DCE Security program interfaces are local client-side APIs only. The ACL facility includes this kind of interface, and some others as well, as follows:

- The DCE ACL interface (**sec_acl**), which enables clients to browse or edit DCE ACLs.
- The DCE ACL manager library (**dce_acl**), which enables servers to perform DCE-conformant authorization checks at runtime. This ACL library provides an implementation of the ACL Manager Interface and the ACL Network Interface. It supports the development of ACL managers for DCE servers.
- The DCE ACL network interface (**rdacl**), which enables servers that manage access control (such as ACL managers that use **sec_acl_mgr**) to communicate with clients that use **sec_acl**.

Figure 106 shows a schematic view of the relationships and usage of these interfaces, as well as some relevant RPC interfaces. This chapter first discusses the client API, and then the two server program interfaces.

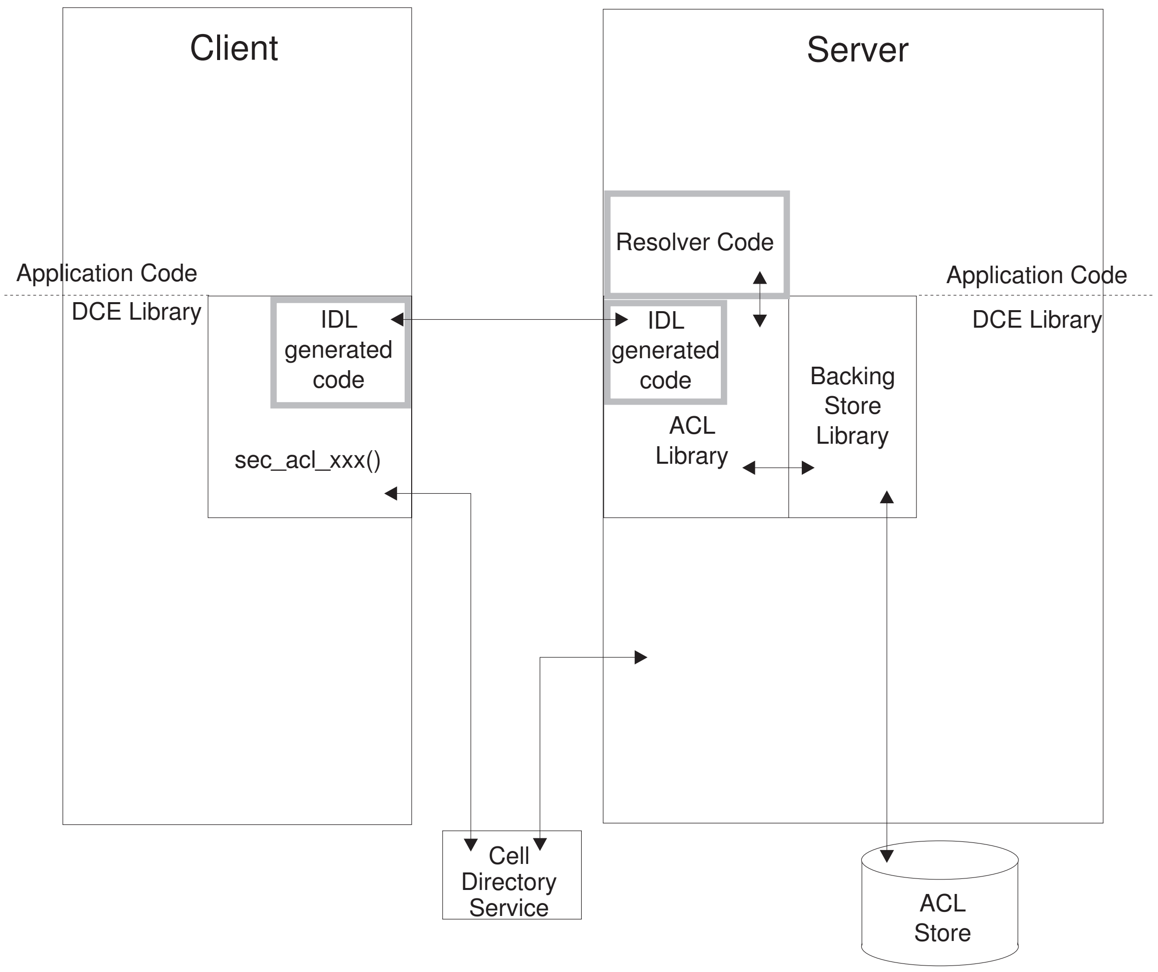


*Figure 106. ACL Program Interfaces*

## The Client-Side API

The client-side API is a local interface consisting of a set of routines that are prefixed **sec_acl**. from which the default DCE ACL editor (**dcecp**) is built. An application that needs to replace **dcecp** with a DCE ACL editor or browser of its own calls this interface. The following sections provide specific information on the functionality that this API supports.

## Binding to an ACL

Any operation performed on an ACL uses an ACL handle to identify the target of the operation. The handle is bound to the object protected by the ACL, not to the ACL itself. Since an object may be protected by more than one ACL manager type, the ACL itself can only be uniquely identified by the ACL handle in combination with the manager type that manages it. ACL editing calls must also specify the ACL type to be read or otherwise manipulated (the object, default container, or default object ACL types).

An application may call **sec_acl_bind()** to get an ACL handle. The handle itself is opaque to the calling program, which needs none of the information encoded in it to use the ACL interface.

A program can obtain a list of ACL manager types protecting an object and pass this data, along with the ACL type identifier, to another client-side routine. There are two calls that perform this function:

| | |
|---|---|
| **sec_acl_get_manager_types()** | Returns a list of UUIDs corresponding to the manager types. |
| **sec_acl_get_mgr_types_semantics()** | Returns a list of UUIDs corresponding to the manager types and the POSIX semantics supported by each manager type. The output of this call is used by the **sec_acl_calc_mask()** routine when it calculates a new **mask_obj** mask. |

(In the absence of a cell naming service, an application may call **sec_acl_bind_to_addr()**; this call binds to a network address rather than a cell namespace entry.)

Once an application is finished using an ACL handle, it may call **sec_acl_release_handle()** to dispose of it.

## ACL Editors and Browsers

After obtaining a handle to the object in question (and using **sec_acl_get_manager_types()** or **sec_acl_get_mgr_types_semantics()** to determine the ACL manager types protecting the object), editors and browsers use the **sec_acl_lookup()** function to return a copy of an object's ACL. Once an object's ACL is loaded in memory, the editor can call **sec_acl_get_printstring()** to receive instructions about how to display the permissions of the ACL in a human-readable form. This call returns a symbol or word for each permission, as well as common combinations of permissions. In addition, the printstring structure includes a short explanation of each permission.

An ACL cannot be changed in part. To change an ACL, an editor must read the entire ACL (the **sec_acl_t** structure), change it, and replace it entirely by calling **sec_acl_replace()**.

If the ACL Manager supports the **mask_obj** mask type, you can use **sec_acl_calc_mask()** to calculate a new **sec_acl_e_type_mask_obj** entry type.

This function is supported for POSIX compatibility only, for those applications that use **mask_obj** with its POSIX semantics. Accordingly, **sec_acl_calc_mask()** returns the union of the permissions of all ACL entries *other than* **user_obj**, **other_obj**, **unauthenticated** (and the preexisting **mask_obj**). These correspond approximately to what POSIX calls the "File Group Class" of ACL entries, although that designation is not appropriate in the DCE context. In particular, **sec_acl_calc_mask()** works independently of the DCE DFS. Use the **sec_acl_get_mgr_types_semantics()** routine to obtain the required POSIX semantics and determine if the manager to which the ACL list will be submitted supports the **sec_acl_e_type_mask_obj** entry type.

An ACL can occupy a substantial amount of memory. The memory management routine, **sec_acl_release()**, frees the memory occupied by an ACL, and returns it to the pool. This is strictly a local operation.

### Testing Access

Access testing by clients is not definitive because the state of an ACL can change between the access test and the request to the server to perform an application operation. More typically, a client simply requests an operation; then, on receiving the request, the server performs the access test, and depending on the result, either runs the client's request or returns an error to the client. However, if an application server acts as a client of another server that manages ACLs for the application objects, the application server needs the results of access tests from the ACL manager server in order to process requests from application clients.

After calling **sec_acl_bind()** to acquire an ACL handle to the target object, such an application server would call **sec_acl_test_access()** with the returned handle, the UUID of the ACL manager, and the permission(s) requested in order to perform the requested operation. The access-test function returns TRUE if the object's ACL allows the client to perform the operation, otherwise it returns FALSE. An alternative to **sec_acl_test_access()** is **sec_acl_get_access()**.

Some applications need to check an ACL on behalf of a principal other than the one represented by the calling process. For example, a replicated database server would presumably need to check the privilege attributes of its clients against the database ACL entries. In this case, the server would use the **sec_acl_test_access_on_behalf()** function, which is identical to the **sec_acl_test_access()** function, except that it also requires the credentials of the principal for which the server principal is acting as an agent.

### Errors

Although the ACL API saves errors received from the DCE RPC runtime (or other APIs) in ACL handle data, it returns an error describing the ACL operation that failed as a result of the RPC error. However, if an error occurs and the client needs to know the cause of the ACL operation failure, it may call **sec_acl_get_error_info()**. This routine returns the error code last stored in the handle.

## Guidelines for Constructing ACL Managers

ACL manager names for all of DCE should follow the convention for naming **dcecp** attributes. There is no architectural restriction involved in the guidelines shown here, merely an attempt at consistency. The **dcecp** program will accept names outside of this convention, but adherence to it will make usage of ACL managers easier.

The guidelines are:

- Alphabetic characters in names must be lower case only.
- Names should not contain underscores.

- Names should not contain spaces.
- Names should be no longer than 16 bytes, the defined value of **sec_acl_printstring_len**.
- Names should be similar to object command names supported in **dcecp** whenever possible. For example, the ACL manager name **principal** refers to the object, **/.:/sec/principal**, that contains registry information about principals. Note that **dcecp** allows abbreviations. For example, a user can specify **org** for the ACL manager name **organization**.
- Names must be unique within a component's ACL manager, but not necessarily within DCE. For example, the name **xattrschema** can be used for a DCED Extended Attribute Configuration Schema ACL object and for a Security Extended Registry Attribute Schema ACL object.
- The helpstring for an ACL manager must specify the component that owns or manages the objects in questions, because this information cannot always be derived from the ACL manager name.

## Extended Naming of Protected Objects

The DCE ACL model supports extended naming, which enables ACL managers to protect separately objects that are not registered in the cell name space. This provides an alternative to registering all the server's objects with CDS. The server alone is registered, and it contains code to identify its own objects by name. To achieve ACL protection for these objects, the ACL manager must be able to identify the ACLs in the same way the server identifies the objects. A resolution routine provides this ability.

Figure 107 shows the example of a printer server that is registered with CDS, with printers that are not. The ACL manager for the printer server uses the **dce_acl_resolve_by_name()** resolution routine to obtain the UUIDs of the several printers that are supported. The administrator in charge of the printers can change the printers, their names, and their ACLs without concern for registering them with CDS.

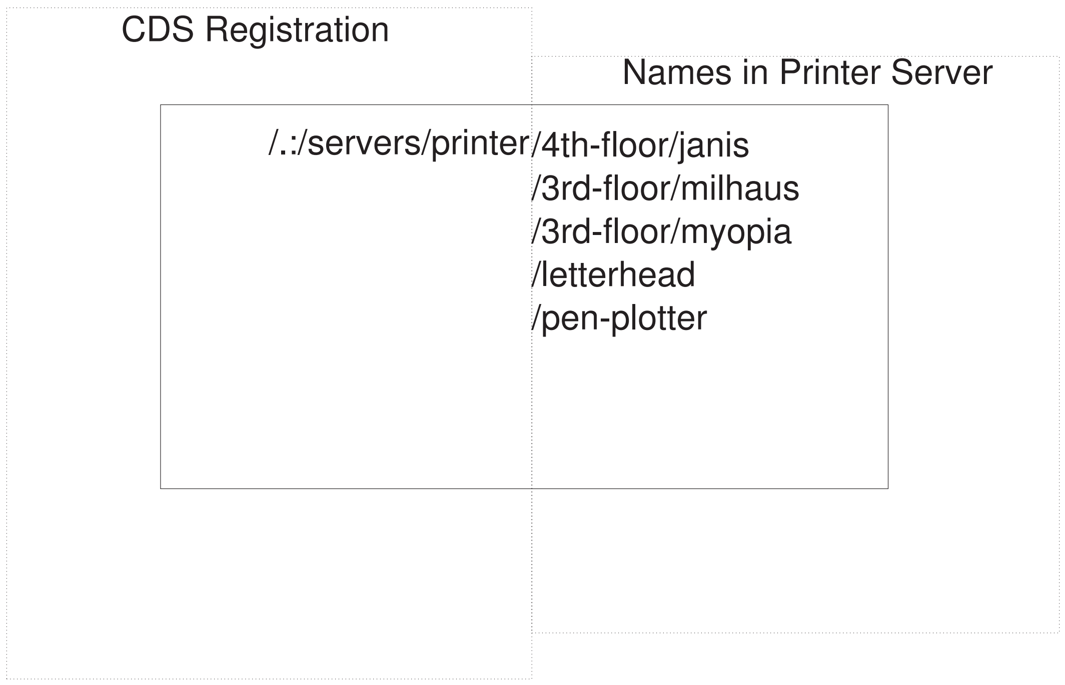


*Figure 107. Protection with Extended Naming*

When the **dce_acl_register_object_type()** routine registers an object type, it associates a resolution routine with the object type. The ACL Library provides two resolution routines,

**dce_acl_resolve_by_name()** and **dce_acl_resolve_by_uuid()**. Other resolution routines can be easily written, as required.

To take advantage of extended naming, an ACL manager must register the server name, object UUID, and **rdaclif.idl** interface with the Cell Directory Service (refer to the *z/OS DCE Application Development Guide: Directory Services* for more information). In addition, the ACL manager must register the object UUID and **rdaclif.idl** interface with the RPC endpoint mapper (refer to Part 2, "Using the DCE Remote Procedure Call APIs" on page 35).

## The ACL Network Interface

The ACL network interface (**rdacl**) provides a DCE-common interface to ACL managers. It is the interface exported by the default DCE ACL managers to the default DCE ACL client (that is, the **dcecp** tool), and any other client that uses the DCE ACL interface (**sec_acl**).

The client API, **sec_acl**, is a local interface that calls a client-side implementation of the ACL network interface. However, you are responsible for writing the server side of this interface. The implementation needs to conform to the sections of the *z/OS DCE Application Development Reference* that describe the **rdacl** routines. Following is a summary of these routines:

| | |
|---|---|
| **rdacl_lookup()** | Retrieves a copy of the object's ACL. |
| **rdacl_replace()** | Replaces the specified ACL. |
| **rdacl_get_access()** | Returns a principal's permissions to an object |
| **rdacl_test_access()** | Determines whether the calling principal has the requested permission(s). |
| **rdacl_test_access_on_behalf()** | Determines whether the principal represented by the calling principal has the requested permission(s). This function returns TRUE if both the principal and the calling principal acting as its agent have the requested permission(s). |
| **rdacl_get_manager_types()** | Returns a list of manager types protecting the object. |
| **rdacl_get_printstring()** | Obtains human-readable representations of permissions. |
| **rdacl_get_referral()** | Returns a referral to an ACL update site. This function enables a client that attempts to change an ACL at a read-only site to recover from the error and rebind to an update site. |

## The ACL Library

The ACL Library provides an implementation of the ACL Manager Interface and the ACL Network Interface for the convenience of programmers who are writing ACL managers for DCE servers.

The ACL Library meets the following needs:

- It provides stable storage for ACLs.
- It implements the **rdacl** interface, including support for multiple object types, initial default object ACLs, and initial default container ACLs.
- It implements the full access algorithm, including masks and delegation.
- It provides DCE developers with a set of convenience functions, so that servers can easily perform common styles of access control with minimal effort.

**ACL Library Capabilities:** The ACL Library provides simple and practical access to the DCE security model.

The library provides a routine that indicates in a single call whether or not a client has the appropriate permissions to perform a particular operation. A server can also easily retrieve the full set of permissions granted to a client by an object's ACL.

The library provides the complete **rdacl** remote interface. Standard routines are provided to map either a UUID attached to a handle or a residual name specified as one of the parameters.

The combination of these capabilities means that most servers will not have any need to use the DCE ACL data types directly.

**The ACL Application Programmer Interface:** The ACL Library API, **dce_acl**, is a local interface that provides the server-side implementation of the ACL network interface. The *z/OS DCE Application Development Reference* describes the library routines.

The ACL library consists of the following parts:

- Initialization routines, where the server registers each ACL manager type.
- Server queries, where a server can perform various types of access checks.
- ACL object creation, where servers can create ACLs without concern for most low-level data type details.
- The **rdacl** implementation and server callback, where the server maps **rdacl** parameters into a specific ACL object. Two sample resolver routines are associated with this part:

| | |
|---|---|
| **dce_acl_resolve_by_name()** | Finds an ACL's UUID, given an object's name. |
| **dce_acl_resolve_by_uuid()** | Finds an ACL's UUID, given an object's UUID. |

***Initialization Routines:*** An ACL manager must first define the types of the objects it manages. For example, a simple directory service would have directories and entries, and each type of object would have a different ACL manager. On a practical level, if a server has different types of objects, then the most common difference between the ACL managers is the printed representation of its permission bits. In other words, although the **sec_acl_printstring_t** values differ, the algorithm for evaluating permissions remains the same.

The ACL library provides a global printstring that specifies the **read**, **write**, and **control** bits. Application developers are encouraged to use this printstring whenever appropriate.

An ACL manager calls the **dce_acl_register_object_type()** routine to register an object type, once for each type of object that the server manages. The manager printstring does not define any permission bits; they are set by the library to be the union of all permissions in the ACL printstring.

The server must register the **rdacl** interface with the RPC runtime and with the endpoint mapper. See the information about **dce_server_register()** in the *z/OS DCE Application Development Reference*.

***Server Queries:*** The ACL library provides several routines to automate the most common use of DCE ACLs:

| | |
|---|---|
| **dce_acl_is_client_authorized()** | Checks whether a client's credentials are authenticated, and if so, that they grant the desired access. |
| **dce_acl_inq_client_permset()** | Returns the client's permissions, corresponding to an ACL. |
| **dce_acl_inq_client_creds()** | Returns the client's credentials. |
| **dce_acl_inq_permset_for_creds()** | Determines a client's complete extent of access to an object. |
| **dce_acl_inq_acl_from_header()** | Retrieves the UUID of an ACL from the header of an object in the backing store. |
| **dce_acl_inq_prin_and_group()** | Inquires the principal and the group of an RPC caller. |

***Creating ACL Objects:*** The following convenience functions may be used by an application programmer to create ACL objects in other servers or clients.

| | |
|---|---|
| **dce_acl_copy_acl()** | Copies an ACL. |
| **dce_acl_obj_init()** | Initializes an ACL for an object. |
| **dce_acl_obj_free_entries()** | Frees space used by an ACL's entries. |
| **dce_acl_obj_add_user_entry()** | Adds permissions for a user ACL entry to the given ACL. |
| **dce_acl_obj_add_group_entry()** | Adds permissions for a group ACL entry to the given ACL. |
| **dce_acl_obj_add_id_entry()** | Adds permissions for an ACL entry to the given ACL. |
| **dce_acl_obj_add_unauth_entry()** | Adds permissions for unauthenticated ACL entry to the given ACL. |
| **dce_acl_obj_add_obj_entry()** | Adds permissions for an “obj” ACL entry to the given ACL. |
| **dce_acl_obj_add_foreign_entry()** | Adds permissions for the ACL entry for a foreign user or group to the given ACL. |
| **dce_acl_obj_add_any_other_entry()** | Adds permissions for the “any_other” ACL entry to a given ACL. |

***RDACL Implementation and Server Callback:*** The ACL Library makes a complete implementation of the **rdacl** interface available to programmers writing servers, in a manner that is mostly transparent to the rest of the server code.

The operations in the **rdacl** interface share an initial set of parameters that specify the ACL object being operated upon:

```
handle_t                     h
sec_acl_component_name_t     component_name
uuid_t                       *manager_type
sec_acl_type_t               sec_acl_type
```

The *sec_acl_type* parameter indicates whether a protection ACL, an initial default object ACL, or an initial default container ACL is desired. It does not appear in the **access** operations as it must have the value **sec_acl_type_object**.

In order to implement the **rdacl** interface, the server must provide a **resolution** routine that maps these parameters into the UUID of the desired ACL object; the library includes two such routines, **dce_acl_resolve_by_uuid()** and **dce_acl_resolve_by_name()**.

The resolution routine is required because servers use the namespace in different ways. Here are three examples:

- Servers that only export their binding information and manage a single object, and hence use a single ACL, do not need the resolution parameters. DTS is an example of this case.
- Servers with many objects in the namespace, with a UUID in each entry, will call **rpc_binding_inq_object** on the handle to obtain the object UUID. They then use this same UUID as the index of the ACL object. Many application servers will be of this type. One ACL Library resolver function, **dce_acl_resolve_by_uuid()**, matches this paradigm. This paradigm is not appropriate if the number of objects is immense.
- Servers with many objects will use a junction or similar architecture so that the component name (also called the **residual**) specifies the ACL object by name. The DCE security server is essentially of this type. Another ACL Library resolver function, **dce_acl_resolve_by_name()**, matches this paradigm.

The following **typedef** specifies the signature for a resolution routine. The first four parameters are the common **rdacl** parameters mentioned above.

```
typedef void (*dce_acl_resolve_func_t)(
/* [in] parameters */
    handle_t                    h,
    sec_acl_component_name_t    component_name,
    sec_acl_type_t              sec_acl_type,
    uuid_t                      *manager_type,
    boolean32                   writing,
    void                        *resolver_arg
/* [out] parameters */
    uuid_t                      *acl_uuid,
    error_status_t              *st
);
```

For situations in which neither of the ACL Library resolver functions, **dce_acl_resolve_by_uuid()** or **dce_acl_resolve_by_name()**, is appropriate, application developers must provide their own.

The following two examples illustrate the general structure of the **dce_acl_resolve_by_uuid()** API and **dce_acl_resolve_by_name()** API that are supplied in the ACL Library. They may be used as paradigms for creating additional resolver routines.

The first example shows **dce_acl_resolve_by_name()**.

A server has several objects, and stores each in a backing store database. Part of the standard header for each object is a structure that contains the UUID of the ACL for that object. (The standard header is not intended to be an abstract type, but rather a common prologue provided to ease server development.) The resolution routine for this server retrieves the object UUID from the handle, uses that as an index into its own backing store, and uses the *sec_acl_type* parameter to retrieve the appropriate ACL UUID from the standard data header.

This routine needs the database handle for the server's object storage, which is specified as the *resolver_arg* parameter in the **dce_acl_register_object_type** call.

```
#define STAT_CHECK_RET(st) { if (st != error_status_ok) return; }
dce_acl_resolve_func_t
dce_acl_resolve_by_uuid(
  /* in */
    handle_t                   h,
    sec_acl_component_name_t   component_name,
    sec_acl_type_t             sec_acl_type,
    uuid_t                     *manager_type,
    boolean32                  writing,
    void                       *resolver_arg,
  /* out */
    uuid_t                     *acl_uuid,
    error_status_t             *st
)
{
    dce_db_handle_t            db_h;
    dce_db_header_t            dbh;
    uuid_t                     obj;

    /* Get the object. */
    rpc_binding_inq_object(h, &obj, st);
    STAT_CHECK_RET(*st);

    /* Get object header using the object backing store.  The handle was
     * passed in as the resolver_arg in the dce_acl_register_object_type call.
     */
    db_h = (dce_db_handle_t)resolver_arg;
    dce_db_std_header_fetch(db_h, &obj, &dbh, st);
    STAT_CHECK_RET(*st);

    /* Get the appropriate ACL based on the ACL type. */
    dce_acl_inq_acl_from_header(dbh, sec_acl_type, acl_uuid, st);
    STAT_CHECK_RET(*st);
}
```

The next example shows **dce_acl_resolve_by_name()**.

A server uses the residual name to resolve an ACL object, by using **dce_acl_resolve_by_name()**. This routine requires a DCE database that maps names into ACL UUIDs. This backing store database must be maintained by the server application so that created objects always get a name, and that name must be a key into a database that stores the UUID identifying the object. The *resolver_arg* parameter given in the **dce_acl_register_object_type** call must be a handle for that database.

```
#define STAT_CHECK_RET(st) { if (st != error_status_ok) return; }
dce_acl_resolve_func_t
dce_acl_resolve_by_name(
  /* in */
    handle_t                   h,
    sec_acl_component_name_t   component_name,
    sec_acl_type_t             sec_acl_type,
    uuid_t                     *manager_type,
    boolean32                  writing,
    void                       *resolver_arg,
  /* out */
    uuid_t                     *acl_uuid,
    error_status_t             *st
)
{
    dce_db_handle_t            db_h;
    dce_db_header_t            dbh;

    /* Get object header using the object backing store.  The handle was
     * passed in as the resolver_arg in the dce_acl_register_object_type call.
     */
    db_h = (dce_db_handle_t)resolver_arg;
    dce_db_std_header_fetch(db_h, component_name, &dbh, st);
    STAT_CHECK_RET(*st);

    /* Get the appropriate ACL based on the ACL type. */
    dce_acl_inq_acl_from_header(dbh, sec_acl_type, acl_uuid, st);
    STAT_CHECK_RET(*st);

}
```

# Chapter 33. The ID Map Application Program Interface

In the multicell environment, the global print string representation of a principal identity can be ambiguous, even though every principal and its local cell have unique names in the form of UUIDs to which the print string representations normally resolve. For example, all ACLs maintain UUIDs as the definitive representations of principal and cell names.

The **acl_edit** tool, on the other hand, takes as input (and also displays) this same information as print strings. This string-to-UUID mapping is accomplished easily when an ACL entry refers to a local identity; that is, a member of the local cell. However, when a user adds an ACL entry for a foreign principal identity such as **/.../world/dce/rd/writers/tom**, it is not evident to the ACL Manager which part of the name identifies the cell, and which identifies the principal within the cell. The name **/... /world/dce** may refer to a cell containing the principal **/rd/writers/tom**, or the cell name may be **/.../world/dce/rd** and the principal name may be **/writers/tom**.

To parse the fully qualified principal name that the user types into its cell name and local principal-name components, and for these components to be mapped to UUIDs, ACL Managers that support entries for foreign identities use the ID Map API. For the same reasons, many other kinds of servers in a DCE multicell environment need a facility to parse global names and translate UUIDs into print string names.

The ID Map API provides a simple interface to translate a fully qualified name (that is, the global representation of a name) into its components and back again. This API consists of the following calls:

| | |
|---|---|
| **sec_id_parse_name()** | Takes as input a registry context handle and a fully qualified principal name, and returns the principal's print string name and UUID, and the print string name and UUID of the principal's local cell. |
| **sec_id_gen_name()** | Translates a principal UUID and the UUID of its local cell UUID into a cell-relative principal name, a cell name, and a fully qualified principal name. |
| **sec_id_parse_group()** | This call is like **sec_id_parse_name()** except that it operates on group names. |
| **sec_id_gen_group()** | This call is like **sec_id_gen_name()** except that it operates on group names. |

# Chapter 34. DCE Audit Service

Audit plays a critical role in distributed systems. Adequate audit facilities are necessary for detecting and recording critical events in distributed applications.

Audit, a key component of DCE, is provided by the **DCE Audit Service**.

This chapter provides an introduction to the DCE Audit Service.

## Features of the DCE Audit Service

The DCE Audit Service has the following features:

- An Audit daemon performs the logging of audit records based on specified criteria.
- Application Programming Interfaces (APIs) can be used as part of application server programs to record audit events. These APIs can also be used to create tools that analyze the audit records.
- An administrative command interface to the Audit daemon directs the daemon in selecting the events that are going to be recorded based on certain criteria.
- An event classification mechanism is used to logically group a set of audit events for ease of administration.
- Audit records can be directed to logs or to the operator console.

## Components of DCE Audit Service

The DCE Audit Service has three basic components:

- Application Programming Interfaces (APIs)

  Provide the functions that are used to detect and record critical events when the application server services a client. The application programmer uses these functions at **code points** in the application server program to actuate the recording of audit events.

  Other APIs are also provided which can be used to create tools that examine and analyze the audit event records.

- Audit daemon (**auditd**)

  Maintains the filters and the central audit logs.

- Audit Management Interface

  Management interface to the Audit daemon. Used by the administrator to specify how the Audit daemon will filter the recording of audit events. This interface is available from the DCE Control Program (**dcecp**).

## DCE Audit Service Concepts

This section briefly describes the DCE Audit concepts that are relevant to DCE application programming.

## Audit Clients

All RPC-based servers are potential audit clients—DCE servers and user-written application servers. The DCE Security Service and Distributed Time Service are auditable. That is, code points (discussed in the next section), are already in place on these services.

The Audit daemon can also audit itself.

## Code Point

A **code point** is a location in the application server program where DCE Audit APIs are used. Code points generally correspond to operations or functions offered by the application server for which audit is required. For example, if a bank server offers the cash withdrawal function **acct_withdraw()**, this function may be deemed to be an auditable event and be designated as a code point.

As mentioned previously, code points are already in place in the DCE Security Service and Distributed Time Service code. The code points for these services are described in the *z/OS DCE Administration Guide*.

## Events

An **audit event** is any event that an audit client wishes to record. Generally, audit events involve the integrity of the system. For example, when a client withdraws cash from his bank account, this can be an audit event.

An audit event is associated with a code point in the application server code.

The terms **audit event**, **event**, and **auditable event** are used interchangeably in this book.

**Event Names and Event Numbers:** Each event has a symbolic name as well as a 32-bit number assigned to it. Symbolic names are used only for documentation in identifying audit events. In creating event classes, the administrator uses the event numbers associated with these events.

Event numbers are 32-bit integers. Each event number is a tuple made up of a **set-ID** and the **event-ID**. The set-ID corresponds to a set of event numbers and is assigned by OSF to an organization or vendor. The event-ID identifies an event within the set of events. The organization or vendor manages the issuance of the event-ID numbers to generate an event number.

Event numbers must be consecutive. That is, within a range of event numbers, no gaps in the consecutive order of the numbers are allowed.

The structure and administration of event numbers can be likened to the structure and administration of IP addresses. Recall that an IP address is a tuple of a network ID (analogous to the set-ID) and a host ID (analogous to the event-ID). The format and administration of event numbers are also analogous to IP addresses, as will be discussed in the next sections.

**Event Number Formats:** Event numbers follow one of five formats (A to E), depending on the number of audit events in the organization. The format of an event number can be determined from its four high-order bits.

- Format A can be used by large organizations (such as OSF or major DCE vendors) that need more than 16 bits for the event-ID. This format allocates 7 bits to the set-ID and 24 bits to the event-ID. Format A event numbers with zero (0) as its set-ID are assigned to OSF. That is, all event numbers used by OSF have a zero in the most significant byte.
- Format B can be used by intermediate-sized organizations that need 8 to 16 bits for the event-ID.
- Format C can be used by small organizations that need less than 8 bits for the event-ID.
- Format D is not administered by OSF and can be used freely within the cell. These event numbers may not be unique across cells and should not be used by application servers that are installed in more than one cell.
- Format E is reserved for future use.

The event number formats are illustrated in Figure 108.

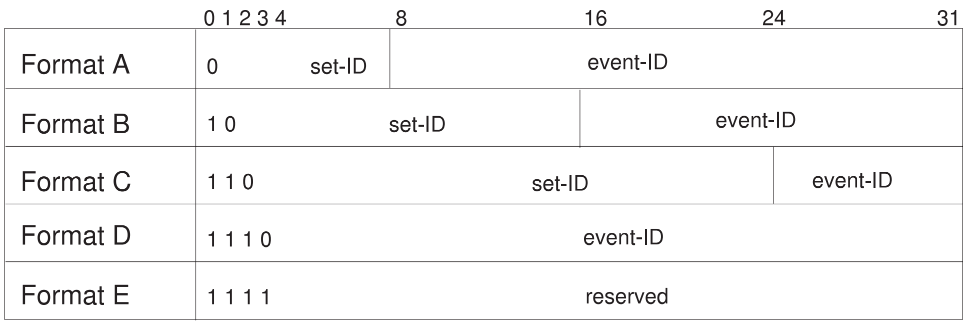


*Figure 108. Event Number Formats*

**Example Event Numbers for DCE Servers:** Following are examples of event numbers in the DCE Security and Time servers, as defined in a header file used by the Security Server and Time Server programs, respectively.

```
/* Event numbers 0x00000100 to 0x000001FF are assigned to the
   Security Server. */

#define AS_Request        0x00000101
#define TGS_TicketReq     0x00000102
#define TGS_RenewReq      0x00000103
#define TGS_ValidateReq   0x00000104
 ...



/* Event numbers 0x00000200 to 0x000002FF are
   assigned to the Time Server. */

#define evt_create_cmd  0x00000200
#define evt_delete_cmd  0x00000201
#define evt_enable_cmd  0x00000202
#define evt_disable_cmd 0x00000203
 ...
```

**Example Event Numbers for an Application Server:** The following is an example of the event numbers in a banking server application, as defined in the application's header file.

```
#define evt_vn_bank_server_acct_open          0x01000000
#define evt_vn_bank_server_acct_close         0x01000001
#define evt_vn_bank_server_acct_withdraw      0x01000002
#define evt_vn_bank_server_acct_deposit       0x01000003
#define evt_vn_bank_server_acct_transfer      0x01000004
```

**Administration of Event Numbers:** Organizations and vendors must administer the event numbers assigned to them (through the set-ID) to maintain the unique assignment of event numbers.

## Event Class

Audit events can be logically grouped together into an **event class**. Event classes provide an efficient mechanism by which sets of events can be specified by a single value. Generally, an event class consists of audit events with some commonality. For example, in a bank server program, the cash transaction events (deposit, withdrawal, and transfer) may be grouped into an event class.

Typically, the administrator creates and maintains event classes. For more details on event classes, see the *z/OS DCE Administration Guide*.

## Event Class Number

Each event class is assigned an **event class number**. Like the event number, the event class number is a 32-bit integer and is administered by OSF. Event class numbers are discussed in more detail in the *z/OS DCE Administration Guide*.

## Filters

Once the code points are identified and placed in the application server, all audit events corresponding to the code points will be logged in the audit trail file, irrespective of the outcome of these audit events. However, recording all audit events under all conditions may neither be practical nor necessary. **Filters** provide a means by which audit records are logged only when certain conditions are satisfied. A filter is composed of **filter guides** that specify these conditions. Filter guides also specify what action to take if the condition (outcome) is met.

A filter answers the following questions:

- Who will be audited?
- What events will be audited?
- What should be the outcome of these events before an audit record is written?
- Will the audit record be logged in the audit trail file, or displayed on the system console, or both?

For example, for the bank server program, you can impose the following conditions before an audit record is written:

> Audit all withdrawal transactions (the audit events) that fail because of access denial (outcome of the event) that are performed by all customers in the DCE cell (who to audit).

**Filter Subject Identity:** A filter is associated with one **filter subject**, which denotes to who the filter applies. The filter subject is the client of the distributed application who caused the event to happen.

For more information on the filter subject identity, see the *z/OS DCE Administration Guide*.

### Audit Records

An audit record has a header and a trailer. The header contains the common information of all events, for example, the identities of the client and the server, group privileges used, address, and time. The trailer contains event-specific information, for example, the dollar amount of a fund-transfer event.

Audit records are initialized and filled by calling the Audit API routines.

There are five stages in the writing of an audit record:

1. First, the audit trail file must be opened.
2. Second, the code point registers an audit event. At this point, the audit record does not yet have any form.
3. The audit record descriptor is built. This is a representation of the audit data which is built by the **dce_aud_start()**, **dce_aud_put_ev_info()**, and **dce_aud_commit()** routines. This is stored in a data structure in the client's core memory until the **dce_aud_commit()** routine is called. This data is not IDL-encoded until the **dce_aud_commit()** call.
4. The audit record is written to the log. This is stored as IDL-encoded data in the audit log.
5. The audit record is transformed into human-readable form and the resulting character string is stored in a buffer by calls to the **dce_aud_next()** and **dce_aud_print()** routines. (This is not an IDL-encoded representation.)

### Audit Trail File

The **audit trail file** contains all the audit records that are written by the Audit daemon or the Audit APIs. You can specify either a central audit trail file or a local audit trail file. The central audit trail file is maintained by the Audit daemon. The local audit trail file is maintained by the Audit APIs. The terms **audit trail file** and **audit trail** are used interchangeably in this book.

## Administration and Programming in DCE Audit

This section gives you an example of how auditing is accomplished using the DCE Audit Service. Both the programmer and the administrator have to perform tasks to enable the writing of audit records in the audit trail. This section looks at the life cycle of an audit trail, from the time that audit events are identified in the server code, to the time that they are filtered and recorded in the audit trail file.

A bank server example illustrates each stage of the life cycle. In this example, the bank server program offers five operations: **acct_open()**, **acct_close()**, **acct_withdraw()**, **acct_deposit()**, and **acct_transfer()**.

## Programmer Tasks

The programmer uses the DCE Audit APIs to enable auditing in the application server program. The following steps show how the programmer performs these tasks.

1. The programmer identifies the code points in the bank server program. Because each of the five operations (corresponding to an RPC interface) offered by the bank server is a security-relevant operation, the programmer deems that all these operations are security-relevant. The programmer assigns a codepoint to each operation.

   ```
   acct_open()        /* first code point */
   ...
   acct_close()       /* second code point */
   ...
   acct_withdraw()    /* third code point */
   ...
   acct_deposit()     /* fourth code point */
   ...
   acct_transfer()    /* fifth code point */
   ...
   ...
   ```

   Each code point corresponds to an audit event.

2. The programmer then assigns an event number to each audit event (corresponding to each code point). For example, define these numbers in the header file as follows:

   ```
     /* event number for the first code point, acct_open() */
   #define evt_vn_bank_server_acct_open      0xC1000000

     /* event number for the second code point, acct_close() */
   #define evt_vn_bank_server_acct_close     0xC1000001

     /* event number for the third code point, acct_withdraw() */
   #define evt_vn_bank_server_acct_withdraw  0xC1000002

     /* event number for the fourth code point, acct_deposit() */
   #define evt_vn_bank_server_acct_deposit    0xC1000003

     /* event number for the fifth code point, acct_transfer() */
   #define evt_vn_bank_server_acct_transfer   0xC1000004
   ```

3. The programmer now starts adding Audit API routines to the bank server program.

   In the initialization part of the server, the application programmer uses the **dce_aud_open()** API to open an audit trail file for writing the audit records. This routine uses the lowest-numbered event as one of its parameters, in this case, 0xC1000000 (evt_vn_bank_server_acct_open). Using the lowest-numbered event enhances the performance of the filter search.

   ```
    ...
    /* open an audit trail file for writing */
    dce_aud_open(aud_c_trl_open_write, description,
                  evt_vn_bank_server_acct_open,
                  5, &audit_trail, &status);
    ...
   ```

4. The programmer invokes the following DCE Audit APIs at each code point :

   - The **dce_aud_start()** API to initialize an audit record. This routine assigns the event number to the event represented by the code point. Thus, it uses the event number corresponding to that code point as one of its parameters.
   - The **dce_aud_put_ev_info()** API to add event-specific information to the audit record. This routine is invoked once for each event-specific item.

- The **dce_aud_commit()** API to commit the audit record in the audit trail file.

The use of these three APIs is illustrated in the following example of the bank server program:

```
    ...
  acct_open()     /* first code point */
  ...
  /* Uses the event number for acct_open(), evt_vn_bank_server_acct_open */
  dce_aud_start(evt_vn_bank_server_acct_open,
                binding,options,outcome,&ard, &status);
 ...
   if (ard) /* If events need to be logged, add trailer info (optional)*/
        dce_aud_put_ev_info(ard,info,&status);
 ...
   if (ard) /* If events need to be logged, add header and trailer info */
        dce_aud_commit(at,ard,options,format,outcome,&status);
 ...
 ...
 ...
  acct_close()   /* second code point */
  ...
  /* Uses the event number for acct_close(), evt_vn_bank_server_acct_close */
  dce_aud_start(evt_vn_bank_server_acct_close,
                binding,options,outcome,&ard, &status);
 ...
   if (ard) /* If events need to be logged */
        dce_aud_put_ev_info(ard,info,&status);
 ...
   if (ard) /* If events need to be logged */
        dce_aud_commit(at,ard,options,format,outcome,&status);
 ...
 ...
 ...
```

5. The programmer uses the **dce_aud_close()** API in the termination routine of the application server. This API closes the audit trail file (and frees up memory) if the application server shuts down.

   The coding of the application program to enable auditing is essentially complete at this point.

## Administrator Tasks

The following steps will be performed by the administrator to filter the audit events and control the audit trail file.

1. The administrator obtains the event numbers corresponding to the events represented by the code points in the bank server program from the programmer or from the program's documentation. These events and their assigned event numbers are:

| | |
|---|---|
| acct_open() | 0xC1000000 |
| acct_close() | 0xC1000001 |
| acct_withdraw() | 0xC1000002 |
| acct_deposit() | 0xC1000003 |
| acct_transfer() | 0xC1000004 |

2. The administrator decides to create two event classes: the **account_creation_operations** class comprised of acct_open() and acct_close(), and the **account_balance_operations** class comprised of acct_withdraw(), acct_deposit(), and acct_transfer().
3. The administrator decides to create two filters, one for all users within the cell (for the cell /.:/torolabcell), and the other for all other users.

The filter for all users within the cell has the following guides:

- Audit the events in the event class **account_balance_operations** only, subject to the next condition.
- Write an audit record only if an operation in that event class failed because of access denial.
- If the first condition is fulfilled, write the audit record in an audit trail file only.

The filter for all other users has the following filter guides:

- Audit the events in both event classes, subject to the next condition.
- Write an audit record if an operation in that event class succeeded or failed.
- Write the audit record both in an audit trail file and the console.

The scenarios described here can be summarized as follows:

- The programmer identifies the code points in the distributed application corresponding to the audit events.
- The programmer uses the Audit API routines on those code points to enable auditing.
- The administrator creates event classes which are used to group the audit events.
- The administrator creates filters to narrow down the conditions by which audit records are written for the audit events. Filters are created with the **dcecp audfilter** command.

Figure 109 illustrates the interactions among the Audit client program, the audit API routines, the Audit daemon (**auditd**), and the Audit management interface (available from **dcecp**).

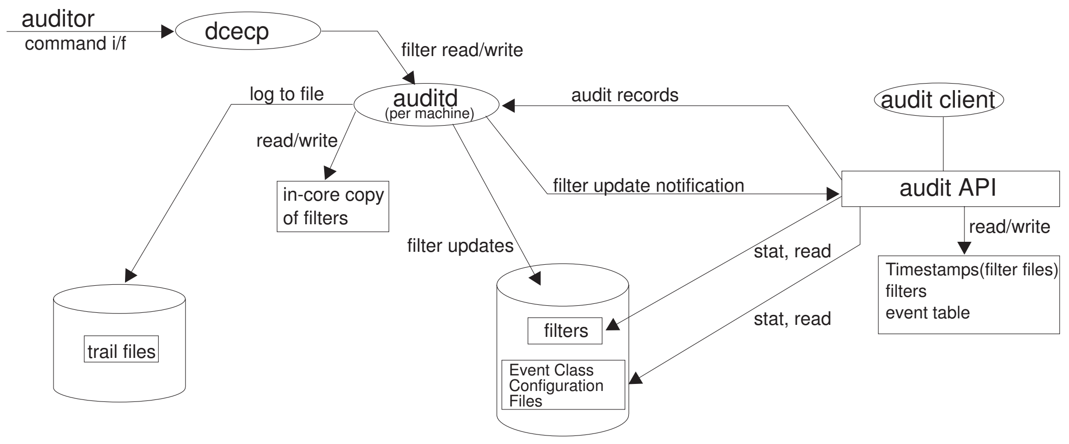


*Figure 109. Overview of the DCE Audit Service*

The Audit management interface (accessed through the DCE Control Program) is used by the systems administrator to specify who, what, when, and how to audit. This is accomplished through the use of the filters. The Audit daemon maintains the filter's information in its address space. The filters are also stored in local files so that the filters can be restored when the machine restarts, and so that Audit clients can read the filter information from these files.

The Audit clients are the users of the filter information. Using the **dce_aud_open** routine, the Audit client reads the information on filters and event class configuration. The Audit client reads these files only once, unless an update notification is received from the Audit daemon (which is triggered by an update initiated by an administrator using the DCE Control program).

# Chapter 35. Using the Audit API Routines

This chapter describes the use of the Audit API routines to add audit capability to distributed applications and to write audit trail analysis and examination tools.

## Adding Audit Capability to Distributed Applications

To record audit events in an audit trail file, the DCE Audit API routines must be called in the distributed application to perform the following:

1. Open the audit trail file during the startup of the application
2. Initialize the audit records at each code point
3. Add Event Information to the audit records at each code point (optional)
4. Commit the audit records at each code point
5. Close the audit trail file when the application shuts down

Note that steps 2, 3, and 4 are repeated in sequence at each code point in the distributed application. Step 3 can be repeated for each event-specific item being put in the audit record.

The use of the Audit API routines in each of these steps is illustrated with the bank server example introduced in Chapter 34, "DCE Audit Service" on page 517.

Five code points are identified in the bank server program: acct_open(), acct_close(), acct_withdraw(), acct_deposit(), and acct_transfer(). Each code point has been assigned an event number and defined in the application server's header file as follows:

```
#define evt_vn_bank_server_acct_open          0x01000000
#define evt_vn_bank_server_acct_close         0x01000001
#define evt_vn_bank_server_acct_withdraw      0x01000002
#define evt_vn_bank_server_acct_deposit       0x01000003
#define evt_vn_bank_server_acct_transfer      0x01000004
```

## Opening the Audit Trail

To open the audit trail file, the main routine of the application server uses the **dce_aud_open** routine. With this routine call, the audit trail file can:

- Be opened for reading or for writing.
- Be directed to the default audit trail file or to a specific file. If **dce_aud_open()** is called without specifying an audit trail file, (by having NULL as the value of the **description** parameter), the default audit trail file is used. This is the **central trail** file which is accessed by RPC calls to the Audit daemon.

  If an audit trail file is specified in the **dce_aud_open()** call (through the **description** parameter), that file is opened directly by the Audit library, bypassing RPCs and the Audit daemon.

In the bank server application, the routine call is:

```
dce_aud_open(aud_c_trl_open_write, &audit_file,
                evt_vn_bank_server_acct_open,
                5, &audit_trail, &status);
```

In this call, the audit trail file **audit_file** is opened for writing. The third parameter **evt_vn_bank_server_acct_open** specifies the lowest event number used in the bank server application.

The fourth parameter, **5**, specifies the number of events defined. The call returns an audit-trail descriptor **audit_trail** that will be used to append audit records to the audit trail file.

## Initializing the Audit Records

Audit records can be initialized by using the **dce_aud_start_*** routines. This routine has five variations, and the use of each variation depends on the available information about the server. In general, if you have the RPC binding information about the server, use the **dce_aud_start()** routine. If not, use the other four variations of this routine, depending on the available information. The five variations are:

| | |
|---|---|
| **dce_aud_start()** | For use by DCE RPC-based server applications. |
| **dce_aud_start_with_server_binding()** | For use by DCE RPC-based client applications |
| **dce_aud_start_with_pac()** | For use by applications that do not use DCE RPC, but use the DCE authorization model. |
| **dce_aud_start_with_name()** | For use by applications that neither use DCE RPC nor the DCE authorization model. |
| **dce_aud_start_with_uuid()** | For use by RPC-based applications that know their client's identity in UUID form. |

The **dce_aud_start_*** routines determine if a specified event must be audited based on the subject identity and event outcome that were defined for that event by the filters.

If the event specifics match the event filters (that is, the event has to be audited), these routines return a pointer to an audit record buffer. If it is determined that the event does not need to be audited, a NULL pointer is returned, and the application can then discontinue any auditing activity. If it cannot be determined whether the event needs to be audited (because the event needs to be audited based on a specific outcome(s) but the outcome is not yet known) these routines return a non-NULL pointer.

When an audit record is initialized, the identification of the audit subject (that is, the client of the distributed application) is recorded.

You can use the **dce_aud_start_*** routines to specify the amount of header information in the audit record. You can specify any or a combination of the following:

- Information on all groups and addresses
- Information on groups only
- Information on addresses only

Using these routines, you can bypass the filter altogether and log the event to the audit trail file or display it on the system console. This option is useful for applications whose events require unconditional audit actions.

In our example, each of the bank server routines (**acct_open()**, **acct_close()**, **acct_withdraw()**, **acct_deposit()**, **acct_transfer()**) will make a **dce_aud_start** routine call. In the **acct_transfer()** routine, the routine call is made as follows:

```
acct_transfer()
...
dce_aud_start (evt_vn_bank_server_acct_transfer,
               h, aud_c_evt_all_info,
               aud_c_esl_cond_success, &ard, &status);
...
```

The **h** points to the RPC binding of the client making the call. The **aud_c_evt_all_info** option means that all information about the client's groups and addresses are included in the audit record header. The **aud_c_esl_cond_success** event outcome means that the event completed successfully.

## Adding Event-Specific Information

If the **dce_aud_start()** routine returns an audit record descriptor to the audit record buffer (meaning that the event needs to be audited), the **dce_aud_put_ev_info()** routine call can be used to add event-specific information to the tail of the audit record.

You can opt not to use the **dce_aud_put_ev_info()** routine if the information provided by the audit record header is already sufficient for your auditing purposes.

If you elect to use this routine, it can be called one or more times, the order of which is preserved in the audit record.

The **dce_aud_put_ev_info()** routine has two parameters: the **ard** parameter, which is the pointer to the audit record descriptor, and the **info** parameter, which is a **dce_aud_ev_info_t** type data containing the event-specific information. The programmer can specify the **dce_aud_ev_info_t** data type to include all the audit information that needs to be collected. For more information on the formats of the audit record, see the *z/OS DCE Application Development Reference*.

In the **acct_transfer()** code point of the bank server example, if you want to record the account numbers of the parties involved in the transfer and the amount of each transaction, the data type declarations and the routine calls can be made as follows:

```
dce_aud_ev_info_t info;

/* account numbers and transfer amounts are all unsigned 32-bit integers */
info.format = aud_c_evt_info_ulong_int;

info.data = acct_from;
dce_aud_put_ev_info(ard, info, &status);
info.data = acct_to;
dce_aud_put_ev_info(ard, info, &status);
info.data = amount;
dce_aud_put_ev_info(ard, info, &status);
```

## Committing an Audit Record

After the header and the optional tail information has been included in the audit record, the **dce_aud_commit()** routine call is used to write the audit record to the audit trail file. This routine uses the audit trail file previously opened by the **dce_aud_open()** routine. You can specify one of two options in the way the routine writes the audit record in the audit trail file:

- Return an error status if the storage or logging service is not available when an attempt is made to write the audit record. This option can be used if the application program can handle write failures in the stable storage.
- If the storage or logging service is not available, keep on trying until the routine is able to write to it. This option can be used if the audit record must be written to stable storage before the routine can proceed safely to another task.

In the bank server example, the routine call can be made as follows:

```
dce_aud_commit(audit_trail, ard, options, format, outcome, &status);
```

The **audit_trail** parameter is the trail descriptor returned in the **dce_aud_open()** call made earlier. The **ard** parameter is the audit record descriptor returned in the **dce_aud_start()** call (and used in the **dce_aud_put_ev_info()** routine call). The **format** parameter specifies a format version number of the event-specific information. The initial version number should be zero, and be incremented when the format changes. For example, the data type used for account numbers might change from 32-bit integer to UUID. The event outcome must be provided in this call, even if it has been provided in the **dce_aud_start()** call made earlier.

### Closing an Audit Trail File

The audit trail file must be closed using the **dce_aud_close()** routine when the application shuts down (because of **rpc_mgmt_stop_server_listening()** routine call or other exceptional conditions). For example, to close the trail, the bank server's main program can make the following routine call:

```
dce_aud_close(audit_trail, &status);
```

This routine flushes buffered audit records to stable storage, and releases the memory allocated for the trail descriptor.

## Writing Audit Trail Analysis and Examination Tools

The Audit APIs can be used to write audit trail analysis and examination tools that selectively review:

- Events that are invoked by one or more subjects, for example, principals, groups, and cells
- Events that have a specific outcome
- Events that occurred during a specified time period
- Events that have specific event IDs

In its most basic form, an audit trail analysis and examination tool must perform five routines:

- Open an audit trail file for reading.
- Read the audit records into a buffer.
- Transform the audit records into human-readable form.
- Discard the audit record.
- Close the audit trail file.

These routines and the APIs that are used for each are discussed in the following sections.

### Opening an Audit Trail File for Reading

To open the audit trail file for reading, use the **dce_aud_open()** routine and specify **aud_c_trl_open_read** as the value for the *flags* parameter. In this case, the values for the *first_evt_number* and *num_of_evts* does not affect the call. For example:

```
    dce_aud_open(aud_c_trl_open_read, AUDIT_TRAIL_FILE,
                 0, 0, &out_trail, status);
```

### Reading the Desired Audit Records into a Buffer

After opening the audit trail file, you can use the **dce_aud_next()** routine to retrieve audit records. Audit records are stored in the audit trail file in binary form. The **dce_aud_next()** routine does not convert the file into readable form. You must use the **dce_aud_print()** routine to translate the audit record into readable form.

The **dce_aud_next()** routine allows you to specify a criteria that will be used in selecting the records that will be read from the file. This criteria is known as **predicates** and is expressed by setting the condition on the value of certain attributes. The condition is set by using any of the following operators: = (equal to), > (greater than), and < (less than).

Predicates can be expressed in any of the following forms:

- *attribute*=*value*
- *attribute*>*value*
- *attribute*= >*value*
- *attribute*<*value*
- *attribute*= <*value*

The following list summarizes these attributes and their acceptable values:

| | |
|---|---|
| SERVER | UUID of the principal which generated the record. |
| EVENT | Audit event number. |
| OUTCOME | Event outcome of the record. |
| STATUS | Authorization status of the application client. |
| CLIENT | UUID of the client principal. |
| TIME | Time when the record was generated. |
| CELL | The UUID of the application client's cell. |
| GROUP | The UUID of the application client's group or groups. |
| ADDR | The address (binding handle) of the client. |

Details of these attributes, their values, and the allowable operators are discussed in the *z/OS DCE Application Development Reference*.

For example, to have the routine retrieve audit records that pertain to a particular outcome only, you can set the predicate to:

```
OUTCOME=SUCCESS
```

If the predicate parameter is set to NULL (that is, no criteria) the next audit record is read. For example, to read the next audit record in a previously opened audit trail file, the following call is made:

```
dce_aud_next(out_trail, NULL, &out_ard, status);
```

You can specify multiple predicates, in which case the predicates are treated as a logical AND condition.

The **dce_aud_next()** routine returns a pointer to the record that was read. This pointer is used by the **dce_aud_print()**, **dce_aud_get_ev_info()**, and **dce_aud_get_header()** routines in transforming the audit records into readable format.

## Transforming the Audit Record into Readable Text

After reading in the desired audit record using the **dce_aud_next()** routine, these binary audit records must be transformed into human-readable form.

You can use any of the following three routines to transform the audit record information to human-readable form:

| | |
|---|---|
| **dce_aud_print()** | Formats the entire audit record (header and tail) into human_readable (ASCII) format. |
| **dce_aud_get_header()** | Obtains the header information of the audit record and formats it into human readable form. |

| | |
|---|---|
| **dce_aud_get_ev_info()** | Obtains the event-specific information in the tail of the audit record and formats it into human-readable form. |

The **dce_aud_next()** routine returns the address of the audit record to these routines. These routines then allocate memory for the buffer (using **malloc()**) and fills it with the human-readable representation of the audit record. The user must call **dce_aud_discard()** to release this memory when all audit record retrieving and transforming tasks have been accomplished. For storage obtained by **dce_aud_get_header**, call **dce_aud_free_header**. For storage obtained by **dce_aud_get_ev_info**, call **dce_aud_free_ev_info**.

## Discarding the Audit Record

The **dce_aud_discard()** routine frees the memory allocated to the binary version of the audit record, that is, the structure returned by the **dce_aud_next()** routine. The **dce_aud_discard()** routine does not free the structures allocated by **dce_aud_print()**, **dce_aud_get_header()**, or **dce_aud_get_ev_info()**.

## Closing the Audit Trail File

Finally, the audit trail file from which the audit records were read must be closed using the **dce_aud_close()** routine.

# Chapter 36. The Password Management Application Programming Interfaces

User passwords are the weakest link in the chain of DCE security. Users, unless their choices are restricted, typically choose passwords that are easy to for them to remember; unfortunately, these memorable passwords are also easy for attackers to “crack.”

The Password Management facility is intended to reduce this risk by providing the tools necessary to develop customized password management servers, and to call them from client password change programs. This facility enables cell administrators to:

- Enforce stricter constraints on users' password choices than those in DCE Standard Policy
- Offer, or force, automatic generation of user passwords

The Password Management facility includes the following APIs:

- The DCE Password Management interface, **sec_pwd_mgmt()**, which enables clients to retrieve a principal's password management ERA values and to request strength-checking and generation of passwords.
- The DCE Password Management network interface, **rsec_pwd_mgmt()**, which enables a Password Management Server to accept and process password strength checking and generation requests.

The following figure provides a schematic view of the relationships and usages of these interfaces, as well as some relevant Security Registry APIs. This chapter first discusses the client API, and then the network API.

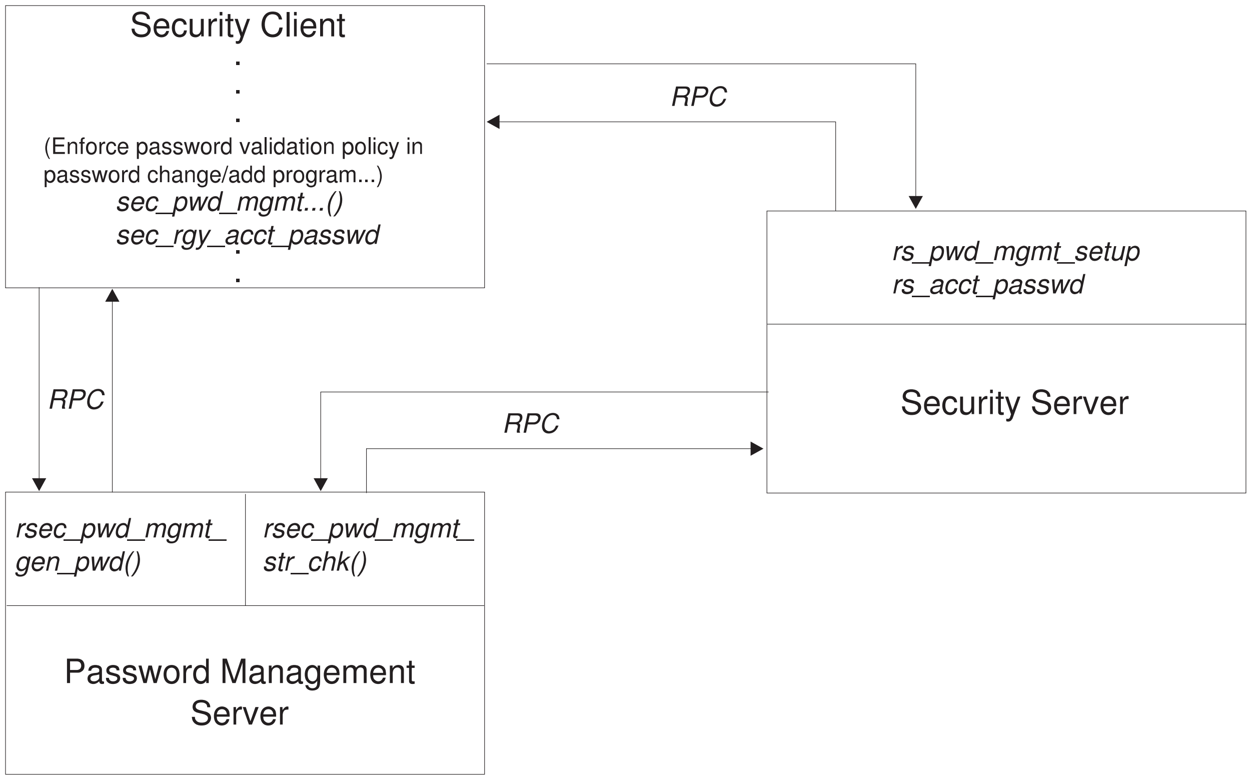


*Figure 110. Usage of Password Management Facility APIs*

For information on how to administer password generation and strength-checking, see the *z/OS DCE Administration Guide*.

## The Client-Side API

**Dcecp** and **rgy_edit** provide support for password generation based on a principal's password validation type ERA. However, if you want to enhance your own password change program, you will need to use the client-side **sec_pwd_mgmt()** API.

This API provides functions that retrieve a principal's password management ERA values, and request password strength-checking and generation from a password management server.

The **sec_pwd_mgmt()** API is defined in the **sec_pwd_mgmt.idl** file.

The general procedure for using the client-side password management API in a password change program is as follows. Refer to Figure 110 on page 531 as you read the following steps:

1. The client calls **sec_pwd_mgmt_setup()**, specifying the login name of the principal whose password is being changed. The Registry Service returns the **pwd_val_type** and **pwd_mgmt_binding** ERAs as well as the Registry standard (password) policy for the principal to the client's security runtime, which is stored in a password management handle (an opaque data type).
2. The client calls **sec_pwd_mgmt_get_val_type()**, specifying the handle returned by **sec_pwd_mgmt_setup()** in step 1. The value of the principal's **pwd_val_type** ERA is extracted from the handle and returned to the client.
3. The client analyzes the principal's **pwd_val_type** ERA to determine whether a generated password is required. If so, it calls **sec_pwd_mgmt_gen_pwd()**, specifying the number of passwords needed, and the handle returned by **sec_pwd_mgmt_setup**. The client security runtime makes an RPC call to the password management server, which generates passwords that adhere to the principal's password policy.
4. The client calls **sec_rgy_acct_passwd()** (or some other form), specifying the new password (either input by the user or generated by **sec_pwd_mgmt_gen_pwd()**). If the principal's **pwd_val_type** ERA mandates it, the Registry Service makes an RPC call to the password management server, specifying the name of the principal and the password to be strength-checked. The password management server checks the format of the password according to the user's password policy and accepts or rejects it.
5. The client calls **sec_pwd_mgmt_free_handle()** to free the memory associated with the password management handle.

## Example of a Password Change Program

Following is an example of a password change program that calls the **sec_pwd_mgmt()** API as described above.

```
sec_pwd_mgmt_setup(&pwd_mgmt_h, context, login_name, login_context, NULL, &st);
if (GOOD_STATUS(&st)) {
    sec_pwd_mgmt_get_val_type(pwd_mgmt_h, &pwd_val_type, &st);
}
if (GOOD_STATUS(&st)) {
    switch (pwd_val_type) {
        case 0: /* NONE */
        case 1: /* USER_SELECT */
            ... get password ...
            break;
        case 2: /* USER_CAN_SELECT */
```

```
            ... if user does not want generated password ... {
                ... get password ...
                break;
            }
        case 3: /* GENERATION_REQUIRED */
            sec_pwd_mgmt_gen_pwd(pwd_mgmt_h, 1, &num_returned,
                &passwd, &st);
            ... display generated password to user - possibly
                prompting for confirmation ...
            break;
    }
}
if (GOOD_STATUS(&st)) {
    sec_rgy_acct_passwd(context, &login_name, &caller_key, &passwd,
        new_keytype, &new_key_version, &st);
}

sec_pwd_mgmt_free_handle(&pwd_mgmt_h, &st);
```

## The Password Management Network Interface

The Password Management interface, **rsec_pwd_mgmt()**, provides a DCE-common interface to Password Management servers. It is the interface exported by the sample password management server provided with DCE1.1 (**pwd_strengthd**) and it is the interface that application developers should use to write their own password management servers. Developers should use the sample code provided as a base for enhancements.

The API is defined in the **rsec_pwd_mgmt.idl** file.

Implementations must conform to the **rsec_pwd_mgmt()** information in the *z/OS DCE Application Development Reference*.

The **rsec_pwd_mgmt()** routines are:

| | |
|---|---|
| **rsec_pwd_mgmt_gen_pwd** | Generates one or more passwords for a given principal. |
| **rsec_pwd_mgmt_str_chk** | Strength checks a principal's password according to policy. |

# Chapter 37. RACF-DCE Interoperability Application Programming Interfaces

This chapter discusses the services available to a z/OS DCE application server to access the DCE information contained in the RACF database. This information can be used when DCE security is used for authentication and RACF is used for authorization to resources.

## DCE APIs

DCE APIs are used to obtain the cell and DCE principal's UUID for DCE 1.1 level clients and servers. Some interfaces are:

- **rpc_binding_inq_auth_caller**, which returns a handle to the client's authorization information
- **sec_cred_get_initiator**, which returns a handle to a single DCE client's authorization information. If delegation has been used, **sec_cred_get_delegate** returns a handle to additional DCE clients' authorization information. This API is issued until no more entries are available.
- **sec_cred_get_pa_data** returns the handle to the Privilege Attribute Certificate (PAC) or Extended Privilege Attribute Certificate (EPAC) that contains the UUIDs of the DCE principal and cell.
- **uuid_to_string** converts the binary form of the UUID to the string form.

For more information on these interfaces, see *z/OS DCE Application Development Reference*.

## z/OS APIs

z/OS UNIX System Services provides two application programming interfaces (APIs) for z/OS DCE application servers to use.

A DCE application server on z/OS can use DCE security services for controlling access to resources that are owned by the application server. As an alternative, the application developer may want to use RACF for controlling access to the set of resources that are managed by the application server. This choice represents a set of tradeoffs. Application servers that use DCE services exclusively on z/OS are the most portable to platforms which support DCE. If portability is not a primary concern, and you want to centralize access control list information in RACF, then non-DCE services such as those described in this section can be considered.

These services enable:

- DCE identity conversion to RACF user ID
- The ability for a DCE server application (with some restrictions) to use RACF services as its ACL (access control list) manager.

They are:

| **Service Name** | **Function** |
|---|---|
| **auth_check_resource_np (BPX1ACK)** | Check access to a RACF protected resource |
| **convert_id_np (BPX1CID)** | Convert a DCE UUID to a user ID or a userid to a DCE UUID |

The BPX1*xxx* names are the names of the kernel callable service that may be used by an application coding to z/OS assembler.

For more information on the use and format of these APIs see *z/OS UNIX System Services Programming: Assembler Callable Services Reference*, SA22-7803.

For the above interfaces, the IBM C library provides C language bindings. The C functions are:

| **C Language Function Name** | **Service** |
|---|---|
| **__check_resource_auth_np()** | Check access to a RACF protected resource |
| **__convert_id_np()** | Convert a DCE UUID to a user ID or a user ID to a DCE UUID |

For more information on the use and format of these APIs see *z/OS C/C++ Run-Time Library Reference*, SA22-7821.

Along with these two APIs, there are a related C Language API, **pthread_security_np()**, and a z/OS UNIX System Services kernel API, **pthread_security_np (BPX1TLS)**. These APIs, along with other thread related services (provided by z/OS UNIX System Services), enable a multithreaded server application to do a unit of work under the RACF identity of a client. Or in other words, the server can become a surrogate for its clients, acting under the identity of its clients.

For more information on the z/OS UNIX System Services kernel **pthread_security_np (BPX1TLS)** API and related thread APIs, see *z/OS UNIX System Services Programming: Assembler Callable Services Reference*, SA22-7803.

# Appendix A. POSIX-based DCE Portable Character Set

*Table 24 (Page 1 of 4). POSIX 1003.2 Portable Character Set (without Control Characters)*

| Symbolic Name | Alternative Name | Character |
|---|---|---|
| <space> | <SP01> | |
| <exclamation-mark> | <SP02> | ! |
| <quotation-mark> | <SP04> | " |
| <number-sign> | <SM01> | # |
| <dollar-sign> | <SC03> | $ |
| <percent-sign> | <SM02> | % |
| <ampersand> | <SM03> | & |
| <apostrophe> | <SP05> | ' |
| <left-parenthesis> | <SP06> | ( |
| <right-parenthesis> | <SP07> | ) |
| <asterisk> | <SM04> | * |
| <plus-sign> | <SA01> | + |
| <comma> | <SP08> | , |
| <hyphen> | <SP10> | - |
| <period> | <SP11> | . |
| <slash> | <SP12> | / |
| <zero> | <ND10> | 0 |
| <one> | <ND01> | 1 |
| <two> | <ND02> | 2 |
| <three> | <ND03> | 3 |
| <four> | <ND04> | 4 |
| <five> | <ND05> | 5 |

*Table 24 (Page 2 of 4). POSIX 1003.2 Portable Character Set (without Control Characters)*

| Symbolic Name | Alternative Name | Character |
|---|---|---|
| <six> | <ND06> | 6 |
| <seven> | <ND07> | 7 |
| <eight> | <ND08> | 8 |
| <nine> | <ND09> | 9 |
| <colon> | <SP13> | : |
| <semicolon> | <SP14> | ; |
| <less-than-sign> | <SA03> | < |
| <equals-sign> | <SA04> | = |
| <greater-than-sign> | <SA05> | > |
| <question-mark> | <SP15> | ? |
| <commercial-at> | <SM05> | @ |
| <A> | <LA02> | A |
| <B> | <LB02> | B |
| <C> | <LC02> | C |
| <D> | <LD02> | D |
| <E> | <LE02> | E |
| <F> | <LF02> | F |
| <G> | <LG02> | G |
| <H> | <LH02> | H |
| <I> | <LI02> | I |
| <J> | <LJ02> | J |
| <K> | <LK02> | K |
| <L> | <LL02> | L |
| <M> | <SM02> | M |
| <N> | <LN02> | N |

*Table 24 (Page 3 of 4). POSIX 1003.2 Portable Character Set (without Control Characters)*

| Symbolic Name | Alternative Name | Character |
|---|---|---|
| <O> | <LO02> | O |
| <P> | <LP02> | P |
| <Q> | <LQ02> | Q |
| <R> | <LR02> | R |
| <S> | <LS02> | S |
| <T> | <LT02> | T |
| <U> | <LU02> | U |
| <V> | <LV02> | V |
| <W> | <LW02> | W |
| <X> | <LX02> | X |
| <Y> | <LY02> | Y |
| <Z> | <LZ02> | Z |
| <left-square-bracket> | <SM06> | [ |
| <backslash> | <SM07> | \ |
| <right-square-bracket> | <SM08> | ] |
| <circumflex> | <SD15> | ^ |
| <underscore> | <SP09> | _ |
| <grave-accent> | <SD13> | ` |
| <a> | <LA01> | a |
| <b> | <LB01> | b |
| <c> | <LC01> | c |
| <d> | <LD01> | d |
| <e> | <LE01> | e |
| <f> | <LF01> | f |
| <g> | <LG01> | g |

*Table 24 (Page 4 of 4). POSIX 1003.2 Portable Character Set (without Control Characters)*

| **Symbolic Name** | **Alternative Name** | **Character** |
|---|---|---|
| <h> | <LH01> | h |
| <i> | <LI01> | i |
| <j> | <LJ01> | j |
| <k> | <LK01> | k |
| <l> | <LL01> | l |
| <m> | <LM01> | m |
| <n> | <LN01> | n |
| <o> | <LO01> | o |
| <p> | <LP01> | p |
| <q> | <LQ01> | q |
| <r> | <LR01> | r |
| <s> | <LS01> | s |
| <t> | <LT01> | t |
| <u> | <LU01> | u |
| <v> | <LU01> | v |
| <w> | <LW01> | w |
| <x> | <LX01> | x |
| <y> | <LY01> | y |
| <z> | <LZ01> | z |
| <left-brace> | <SM11> | { |
| <vertical-line> | <SM13> | \| |
| <right-brace> | <SM14> | } |
| <tilde> | <SD19> | ~ |

# Appendix B. IBM Code Pages

## Code Page IBM-1027

| HEX DIGITS 1ST → 2ND ↓ | 4- | 5- | 6- | 7- | 8- | 9- | A- | B- | C- | D- | E- | F- |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| -0 | (SP) SP010000 | & SM030000 | - SP100000 | ｺ JK500000 | | | ‾ SM150000 | ^ SD150000 | { SM110000 | } SM140000 | \ SM070000 | 0 ND100000 |
| -1 | | ｩ JU010000 | / SP120000 | ｻ JS100000 | a LA010000 | j LJ010000 | ~ SD190000 | £ SC020000 | A LA020000 | J LJ020000 | | 1 ND010000 |
| -2 | ｡ JQ700000 | ｪ JE010000 | ｲ JI000000 | ｼ JS200000 | b LB010000 | k LK010000 | s LS010000 | ¥ SC050000 | B LB020000 | K LK020000 | S LS020000 | 2 ND020000 |
| -3 | ｢ JQ710000 | ｫ JO010000 | ｳ JU000000 | ｽ JS300000 | c LC010000 | l LL010000 | t LT010000 | ﾔ JY100000 | C LC020000 | L LL020000 | T LT020000 | 3 ND030000 |
| -4 | ｣ JQ720000 | ｬ JY110000 | ｴ JE000000 | ｾ JS400000 | d LD010000 | m LM010000 | u LU010000 | ﾕ JY300000 | D LD020000 | M LM020000 | U LU020000 | 4 ND040000 |
| -5 | ､ JQ730000 | ｭ JY310000 | ｵ JO000000 | ｿ JS500000 | e LE010000 | n LN010000 | v LV010000 | ﾖ JY500000 | E LE020000 | N LN020000 | V LV020000 | 5 ND050000 |
| -6 | ･ JQ740000 | ｮ JY510000 | ｶ JK100000 | ﾀ JT100000 | f LF010000 | o LO010000 | w LW010000 | ﾗ JR100000 | F LF020000 | O LO020000 | W LW020000 | 6 ND060000 |
| -7 | ｦ JW500000 | ｯ JT310000 | ｷ JK200000 | ﾁ JT200000 | g LG010000 | p LP010000 | x LX010000 | ﾘ JR200000 | G LG020000 | P LP020000 | X LX020000 | 7 ND070000 |
| -8 | ｧ JA010000 | ｰ JX700000 | ｸ JK300000 | ﾂ JT300000 | h LH010000 | q LQ010000 | y LY010000 | ﾙ JR300000 | H LH020000 | Q LQ020000 | Y LY020000 | 8 ND080000 |
| -9 | ｨ JI010000 | ｱ JA000000 | ｹ JK400000 | ` SD130000 | i LI010000 | r LR010000 | z LZ010000 | ﾚ JR400000 | I LI020000 | R LR020000 | Z LZ020000 | 9 ND090000 |
| -A | ¢ SC040000 | ! SP020000 | | : SP130000 | ﾃ JT400000 | ﾉ JN500000 | ﾏ JM100000 | ﾛ JR500000 | | | | |
| -B | . SP110000 | $ SC030000 | , SP080000 | # SM010000 | ﾄ JT500000 | ﾊ JH100000 | ﾐ JM200000 | ﾜ JW100000 | | | | |
| -C | < SA030000 | * SM040000 | % SM020000 | @ SM050000 | ﾅ JN100000 | ﾋ JH200000 | ﾑ JM300000 | ﾝ JN000000 | | | | |
| -D | ( SP060000 | ) SP070000 | _ SP090000 | ' SP050000 | ﾆ JN200000 | ﾌ JH300000 | [ SM060000 | ] SM080000 | | | | |
| -E | + SA010000 | ; SP140000 | > SA050000 | = SA040000 | ﾇ JN300000 | ﾍ JH400000 | ﾒ JM400000 | ﾞ JX710000 | | | | |
| -F | \| SM130000 | ¬ SM660000 | ? SP150000 | " SP040000 | ﾈ JN400000 | ﾎ JH500000 | ﾓ JM500000 | ﾟ JX720000 | | | | (EO) |

**Code Page 01027**

*Figure 111. Code Page IBM-1027*

## Code Page IBM-1047

| HEX DIGITS 1ST → 2ND ↓ | 4- | 5- | 6- | 7- | 8- | 9- | A- | B- | C- | D- | E- | F- |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| -0 | (SP) SP010000 | & SM030000 | - SP100000 | ø LO610000 | Ø LO620000 | ° SM190000 | µ SM170000 | ¬ SM660000 | { SM110000 | } SM140000 | \ SM070000 | 0 ND100000 |
| -1 | (RSP) SP300000 | é LE110000 | / SP120000 | É LE120000 | a LA010000 | j LJ010000 | ~ SD190000 | £ SC020000 | A LA020000 | J LJ020000 | ÷ SA060000 | 1 ND010000 |
| -2 | â LA150000 | ê LE150000 | Â LA160000 | Ê LE160000 | b LB010000 | k LK010000 | s LS010000 | ¥ SC050000 | B LB020000 | K LK020000 | S LS020000 | 2 ND020000 |
| -3 | ä LA170000 | ë LE170000 | Ä LA180000 | Ë LE180000 | c LC010000 | l LL010000 | t LT010000 | · SD630000 | C LC020000 | L LL020000 | T LT020000 | 3 ND030000 |
| -4 | à LA130000 | è LE130000 | À LA140000 | È LE140000 | d LD010000 | m LM010000 | u LU010000 | © SM520000 | D LD020000 | M LM020000 | U LU020000 | 4 ND040000 |
| -5 | á LA110000 | í LI110000 | Á LA120000 | Í LI120000 | e LE010000 | n LN010000 | v LV010000 | § SM240000 | E LE020000 | N LN020000 | V LV020000 | 5 ND050000 |
| -6 | ã LA190000 | î LI150000 | Ã LA200000 | Î LI160000 | f LF010000 | o LO010000 | w LW010000 | ¶ SM250000 | F LF020000 | O LO020000 | W LW020000 | 6 ND060000 |
| -7 | å LA270000 | ï LI170000 | Å LA280000 | Ï LI180000 | g LG010000 | p LP010000 | x LX010000 | ¼ NF040000 | G LG020000 | P LP020000 | X LX020000 | 7 ND070000 |
| -8 | ç LC410000 | ì LI130000 | Ç LC420000 | Ì LI140000 | h LH010000 | q LQ010000 | y LY010000 | ½ NF010000 | H LH020000 | Q LQ020000 | Y LY020000 | 8 ND080000 |
| -9 | ñ LN190000 | ß LS610000 | Ñ LN200000 | ` SD130000 | i LI010000 | r LR010000 | z LZ010000 | ¾ NF050000 | I LI020000 | R LR020000 | Z LZ020000 | 9 ND090000 |
| -A | ¢ SC040000 | ! SP020000 | ¦ SM650000 | : SP130000 | « SP170000 | ª SM210000 | ¡ SP030000 | Ý LY120000 | (SHY) SP320000 | ¹ ND011000 | ² ND021000 | ³ ND031000 |
| -B | . SP110000 | $ SC030000 | , SP080000 | # SM010000 | » SP180000 | º SM200000 | ¿ SP160000 | ¨ SD170000 | ô LO150000 | û LU150000 | Ô LO160000 | Û LU160000 |
| -C | < SA030000 | * SM040000 | % SM020000 | @ SM050000 | ð LD630000 | æ LA510000 | Ð LD620000 | ¯ SM150000 | ö LO170000 | ü LU170000 | Ö LO180000 | Ü LU180000 |
| -D | ( SP060000 | ) SP070000 | _ SP090000 | ' SP050000 | ý LY110000 | ¸ SD410000 | [ SM060000 | ] SM080000 | ò LO130000 | ù LU130000 | Ò LO140000 | Ù LU140000 |
| -E | + SA010000 | ; SP140000 | > SA050000 | = SA040000 | þ LT630000 | Æ LA520000 | Þ LT640000 | ´ SD110000 | ó LO110000 | ú LU110000 | Ó LO120000 | Ú LU120000 |
| -F | \| SM130000 | ^ SD150000 | ? SP150000 | " SP040000 | ± SA020000 | ¤ SC010000 | ® SM530000 | × SA070000 | õ LO190000 | ÿ LY170000 | Õ LO200000 | (EO) |

**Code Page 01047**

*Figure 112. Code Page IBM-1047*

## Code Page IBM-037

| HEX DIGITS 1ST → 2ND ↓ | 4- | 5- | 6- | 7- | 8- | 9- | A- | B- | C- | D- | E- | F- |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| -0 | (SP) SP010000 | & SM030000 | - SP100000 | ø LO610000 | Ø LO620000 | ° SM190000 | µ SM170000 | ^ SD150000 | { SM110000 | } SM140000 | \ SM070000 | 0 ND100000 |
| -1 | (RSP) SP300000 | é LE110000 | / SP120000 | É LE120000 | a LA010000 | j LJ010000 | ~ SD190000 | £ SC020000 | A LA020000 | J LJ020000 | ÷ SA060000 | 1 ND010000 |
| -2 | â LA150000 | ê LE150000 | Â LA160000 | Ê LE160000 | b LB010000 | k LK010000 | s LS010000 | ¥ SC050000 | B LB020000 | K LK020000 | S LS020000 | 2 ND020000 |
| -3 | ä LA170000 | ë LE170000 | Ä LA180000 | Ë LE180000 | c LC010000 | l LL010000 | t LT010000 | · SD630000 | C LC020000 | L LL020000 | T LT020000 | 3 ND030000 |
| -4 | à LA130000 | è LE130000 | À LA140000 | È LE140000 | d LD010000 | m LM010000 | u LU010000 | © SM520000 | D LD020000 | M LM020000 | U LU020000 | 4 ND040000 |
| -5 | á LA110000 | í LI110000 | Á LA120000 | Í LI120000 | e LE010000 | n LN010000 | v LV010000 | § SM240000 | E LE020000 | N LN020000 | V LV020000 | 5 ND050000 |
| -6 | ã LA190000 | î LI150000 | Ã LA200000 | Î LI160000 | f LF010000 | o LO010000 | w LW010000 | ¶ SM250000 | F LF020000 | O LO020000 | W LW020000 | 6 ND060000 |
| -7 | å LA270000 | ï LI170000 | Å LA280000 | Ï LI180000 | g LG010000 | p LP010000 | x LX010000 | ¼ NF040000 | G LG020000 | P LP020000 | X LX020000 | 7 ND070000 |
| -8 | ç LC410000 | ì LI130000 | Ç LC420000 | Ì LI140000 | h LH010000 | q LQ010000 | y LY010000 | ½ NF010000 | H LH020000 | Q LQ020000 | Y LY020000 | 8 ND080000 |
| -9 | ñ LN190000 | ß LS610000 | Ñ LN200000 | ` SD130000 | i LI010000 | r LR010000 | z LZ010000 | ¾ NF050000 | I LI020000 | R LR020000 | Z LZ020000 | 9 ND090000 |
| -A | ¢ SC040000 | ! SP020000 | ¦ SM650000 | : SP130000 | « SP170000 | ª SM210000 | ¡ SP030000 | [ SM060000 | (SHY) SP320000 | $^1$ ND011000 | $^2$ ND021000 | $^3$ ND031000 |
| -B | . SP110000 | $ SC030000 | , SP080000 | # SM010000 | » SP180000 | º SM200000 | ¿ SP160000 | ] SM080000 | ô LO150000 | û LU150000 | Ô LO160000 | Û LU160000 |
| -C | < SA030000 | * SM040000 | % SM020000 | @ SM050000 | ð LD630000 | æ LA510000 | Ð LD620000 | ¯ SM150000 | ö LO170000 | ü LU170000 | Ö LO180000 | Ü LU180000 |
| -D | ( SP060000 | ) SP070000 | _ SP090000 | ' SP050000 | ý LY110000 | ¸ SD410000 | Ý LY120000 | ¨ SD170000 | ò LO130000 | ù LU130000 | Ò LO140000 | Ù LU140000 |
| -E | + SA010000 | ; SP140000 | > SA050000 | = SA040000 | þ LT630000 | Æ LA520000 | Þ LT640000 | ´ SD110000 | ó LO110000 | ú LU110000 | Ó LO120000 | Ú LU120000 |
| -F | \| SM130000 | ¬ SM660000 | ? SP150000 | " SP040000 | ± SA020000 | ¤ SC010000 | ® SM530000 | × SA070000 | õ LO190000 | ÿ LY170000 | Õ LO200000 | (EO) |

**Code Page 00037**

*Figure 113. Code Page IBM-037*

## Code Page IBM-273

| HEX DIGITS 1ST → 2ND ↓ | 4- | 5- | 6- | 7- | 8- | 9- | A- | B- | C- | D- | E- | F- |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| -0 | (SP) SP010000 | & SM030000 | - SP100000 | ø LO610000 | Ø LO620000 | ° SM190000 | µ SM170000 | ¢ SC040000 | ä LA170000 | ü LU170000 | Ö LO180000 | 0 ND100000 |
| -1 | (RSP) SP300000 | é LE110000 | / SP120000 | É LE120000 | a LA010000 | j LJ010000 | ß LS610000 | £ SC020000 | A LA020000 | J LJ020000 | ÷ SA060000 | 1 ND010000 |
| -2 | â LA150000 | ê LE150000 | Â LA160000 | Ê LE160000 | b LB010000 | k LK010000 | s LS010000 | ¥ SC050000 | B LB020000 | K LK020000 | S LS020000 | 2 ND020000 |
| -3 | { SM110000 | ë LE170000 | [ SM060000 | Ë LE180000 | c LC010000 | l LL010000 | t LT010000 | · SD630000 | C LC020000 | L LL020000 | T LT020000 | 3 ND030000 |
| -4 | à LA130000 | è LE130000 | À LA140000 | È LE140000 | d LD010000 | m LM010000 | u LU010000 | © SM520000 | D LD020000 | M LM020000 | U LU020000 | 4 ND040000 |
| -5 | á LA110000 | í LI110000 | Á LA120000 | Í LI120000 | e LE010000 | n LN010000 | v LV010000 | @ SM050000 | E LE020000 | N LN020000 | V LV020000 | 5 ND050000 |
| -6 | ã LA190000 | î LI150000 | Ã LA200000 | Î LI160000 | f LF010000 | o LO010000 | w LW010000 | ¶ SM250000 | F LF020000 | O LO020000 | W LW020000 | 6 ND060000 |
| -7 | å LA270000 | ï LI170000 | Å LA280000 | Ï LI180000 | g LG010000 | p LP010000 | x LX010000 | ¼ NF040000 | G LG020000 | P LP020000 | X LX020000 | 7 ND070000 |
| -8 | ç LC410000 | ì LI130000 | Ç LC420000 | Ì LI140000 | h LH010000 | q LQ010000 | y LY010000 | ½ NF010000 | H LH020000 | Q LQ020000 | Y LY020000 | 8 ND080000 |
| -9 | ñ LN190000 | ~ SD190000 | Ñ LN200000 | ` SD130000 | i LI010000 | r LR010000 | z LZ010000 | ¾ NF050000 | I LI020000 | R LR020000 | Z LZ020000 | 9 ND090000 |
| -A | Ä LA180000 | Ü LU180000 | ö LO170000 | : SP130000 | « SP170000 | ª SM210000 | ¡ SP030000 | ¬ SM660000 | (SHY) SP320000 | ¹ ND011000 | ² ND021000 | ³ ND031000 |
| -B | . SP110000 | $ SC030000 | , SP080000 | # SM010000 | » SP180000 | º SM200000 | ¿ SP160000 | \| SM130000 | ô LO150000 | û LU150000 | Ô LO160000 | Û LU160000 |
| -C | < SA030000 | * SM040000 | % SM020000 | § SM240000 | ð LD630000 | æ LA510000 | Đ LD620000 | ¯ SM150000 | ¦ SM650000 | } SM140000 | \ SM070000 | ] SM080000 |
| -D | ( SP060000 | ) SP070000 | _ SP090000 | ' SP050000 | ý LY110000 | ¸ SD410000 | Ý LY120000 | ¨ SD170000 | ò LO130000 | ù LU130000 | Ò LO140000 | Ù LU140000 |
| -E | + SA010000 | ; SP140000 | > SA050000 | = SA040000 | þ LT630000 | Æ LA520000 | Þ LT640000 | ´ SD110000 | ó LO110000 | ú LU110000 | Ó LO120000 | Ú LU120000 |
| -F | ! SP020000 | ^ SD150000 | ? SP150000 | " SP040000 | ± SA020000 | ¤ SC010000 | ® SM530000 | × SA070000 | õ LO190000 | ÿ LY170000 | Õ LO200000 | (EO) |

**Code Page 00273**

*Figure 114. Code Page IBM-273*

## Code Page IBM-277

| HEX DIGITS 1ST → 2ND ↓ | 4- | 5- | 6- | 7- | 8- | 9- | A- | B- | C- | D- | E- | F- |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| -0 | (SP) SP010000 | & SM030000 | - SP100000 | ¦ SM650000 | @ SM050000 | ° SM190000 | µ SM170000 | ¢ SC040000 | æ LA510000 | å LA270000 | \ SM070000 | 0 ND100000 |
| -1 | (RSP) SP300000 | é LE110000 | / SP120000 | É LE120000 | a LA010000 | j LJ010000 | ü LU170000 | £ SC020000 | A LA020000 | J LJ020000 | ÷ SA060000 | 1 ND010000 |
| -2 | â LA150000 | ê LE150000 | Â LA160000 | Ê LE160000 | b LB010000 | k LK010000 | s LS010000 | ¥ SC050000 | B LB020000 | K LK020000 | S LS020000 | 2 ND020000 |
| -3 | ä LA170000 | ë LE170000 | Ä LA180000 | Ë LE180000 | c LC010000 | l LL010000 | t LT010000 | · SD630000 | C LC020000 | L LL020000 | T LT020000 | 3 ND030000 |
| -4 | à LA130000 | è LE130000 | À LA140000 | È LE140000 | d LD010000 | m LM010000 | u LU010000 | © SM520000 | D LD020000 | M LM020000 | U LU020000 | 4 ND040000 |
| -5 | á LA110000 | í LI110000 | Á LA120000 | Í LI120000 | e LE010000 | n LN010000 | v LV010000 | § SM240000 | E LE020000 | N LN020000 | V LV020000 | 5 ND050000 |
| -6 | ã LA190000 | î LI150000 | Ã LA200000 | Î LI160000 | f LF010000 | o LO010000 | w LW010000 | ¶ SM250000 | F LF020000 | O LO020000 | W LW020000 | 6 ND060000 |
| -7 | } SM140000 | ï LI170000 | $ SC030000 | Ï LI180000 | g LG010000 | p LP010000 | x LX010000 | ¼ NF040000 | G LG020000 | P LP020000 | X LX020000 | 7 ND070000 |
| -8 | ç LC410000 | ì LI130000 | Ç LC420000 | Ì LI140000 | h LH010000 | q LQ010000 | y LY010000 | ½ NF010000 | H LH020000 | Q LQ020000 | Y LY020000 | 8 ND080000 |
| -9 | ñ LN190000 | ß LS610000 | Ñ LN200000 | ` SD130000 | i LI010000 | r LR010000 | z LZ010000 | ¾ NF050000 | I LI020000 | R LR020000 | Z LZ020000 | 9 ND090000 |
| -A | # SM010000 | ¤ SC010000 | ø LO610000 | : SP130000 | « SP170000 | ª SM210000 | ¡ SP030000 | ¬ SM660000 | (SHY) SP320000 | ¹ ND011000 | ² ND021000 | ³ ND031000 |
| -B | . SP110000 | Å LA280000 | , SP080000 | Æ LA520000 | » SP180000 | º SM200000 | ¿ SP160000 | \| SM130000 | ô LO150000 | û LU150000 | Ô LO160000 | Û LU160000 |
| -C | < SA030000 | * SM040000 | % SM020000 | Ø LO620000 | ð LD630000 | { SM110000 | Đ LD620000 | ¯ SM150000 | ö LO170000 | ~ SD190000 | Ö LO180000 | Ü LU180000 |
| -D | ( SP060000 | ) SP070000 | _ SP090000 | ' SP050000 | ý LY110000 | ¸ SD410000 | Ý LY120000 | ¨ SD170000 | ò LO130000 | ù LU130000 | Ò LO140000 | Ù LU140000 |
| -E | + SA010000 | ; SP140000 | > SA050000 | = SA040000 | þ LT630000 | [ SM060000 | Þ LT640000 | ´ SD110000 | ó LO110000 | ú LU110000 | Ó LO120000 | Ú LU120000 |
| -F | ! SP020000 | ^ SD150000 | ? SP150000 | " SP040000 | ± SA020000 | ] SM080000 | ® SM530000 | × SA070000 | õ LO190000 | ÿ LY170000 | Õ LO200000 | (EO) |

**Code Page 00277**

*Figure 115. Code Page IBM-277*

## Code Page IBM-278

| HEX DIGITS 1ST → 2ND ↓ | 4- | 5- | 6- | 7- | 8- | 9- | A- | B- | C- | D- | E- | F- |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| -0 | (SP)<br>SP010000 | &<br>SM030000 | -<br>SP100000 | ø<br>LO610000 | Ø<br>LO620000 | °<br>SM190000 | µ<br>SM170000 | ¢<br>SC040000 | ä<br>LA170000 | å<br>LA270000 | É<br>LE120000 | 0<br>ND100000 |
| -1 | (RSP)<br>SP300000 | `<br>SD130000 | /<br>SP120000 | \<br>SM070000 | a<br>LA010000 | j<br>LJ010000 | ü<br>LU170000 | £<br>SC020000 | A<br>LA020000 | J<br>LJ020000 | ÷<br>SA060000 | 1<br>ND010000 |
| -2 | â<br>LA150000 | ê<br>LE150000 | Â<br>LA160000 | Ê<br>LE160000 | b<br>LB010000 | k<br>LK010000 | s<br>LS010000 | ¥<br>SC050000 | B<br>LB020000 | K<br>LK020000 | S<br>LS020000 | 2<br>ND020000 |
| -3 | {<br>SM110000 | ë<br>LE170000 | #<br>SM010000 | Ë<br>LE180000 | c<br>LC010000 | l<br>LL010000 | t<br>LT010000 | ·<br>SD630000 | C<br>LC020000 | L<br>LL020000 | T<br>LT020000 | 3<br>ND030000 |
| -4 | à<br>LA130000 | è<br>LE130000 | À<br>LA140000 | È<br>LE140000 | d<br>LD010000 | m<br>LM010000 | u<br>LU010000 | ©<br>SM520000 | D<br>LD020000 | M<br>LM020000 | U<br>LU020000 | 4<br>ND040000 |
| -5 | á<br>LA110000 | í<br>LI110000 | Á<br>LA120000 | Í<br>LI120000 | e<br>LE010000 | n<br>LN010000 | v<br>LV010000 | [<br>SM060000 | E<br>LE020000 | N<br>LN020000 | V<br>LV020000 | 5<br>ND050000 |
| -6 | ã<br>LA190000 | î<br>LI150000 | Ã<br>LA200000 | Î<br>LI160000 | f<br>LF010000 | o<br>LO010000 | w<br>LW010000 | ¶<br>SM250000 | F<br>LF020000 | O<br>LO020000 | W<br>LW020000 | 6<br>ND060000 |
| -7 | }<br>SM140000 | ï<br>LI170000 | $<br>SC030000 | Ï<br>LI180000 | g<br>LG010000 | p<br>LP010000 | x<br>LX010000 | ¼<br>NF040000 | G<br>LG020000 | P<br>LP020000 | X<br>LX020000 | 7<br>ND070000 |
| -8 | ç<br>LC410000 | ì<br>LI130000 | Ç<br>LC420000 | Ì<br>LI140000 | h<br>LH010000 | q<br>LQ010000 | y<br>LY010000 | ½<br>NF010000 | H<br>LH020000 | Q<br>LQ020000 | Y<br>LY020000 | 8<br>ND080000 |
| -9 | ñ<br>LN190000 | ß<br>LS610000 | Ñ<br>LN200000 | é<br>LE110000 | i<br>LI010000 | r<br>LR010000 | z<br>LZ010000 | ¾<br>NF050000 | I<br>LI020000 | R<br>LR020000 | Z<br>LZ020000 | 9<br>ND090000 |
| -A | §<br>SM240000 | ¤<br>SC010000 | ö<br>LO170000 | :<br>SP130000 | «<br>SP170000 | ª<br>SM210000 | ¡<br>SP030000 | ¬<br>SM660000 | (SHY)<br>SP320000 | ¹<br>ND011000 | ²<br>ND021000 | ³<br>ND031000 |
| -B | .<br>SP110000 | Å<br>LA280000 | ,<br>SP080000 | Ä<br>LA180000 | »<br>SP180000 | º<br>SM200000 | ¿<br>SP160000 | \|<br>SM130000 | ô<br>LO150000 | û<br>LU150000 | Ô<br>LO160000 | Û<br>LU160000 |
| -C | <<br>SA030000 | *<br>SM040000 | %<br>SM020000 | Ö<br>LO180000 | ð<br>LD630000 | æ<br>LA510000 | Ð<br>LD620000 | ¯<br>SM150000 | ¦<br>SM650000 | ~<br>SD190000 | @<br>SM050000 | Ü<br>LU180000 |
| -D | (<br>SP060000 | )<br>SP070000 | _<br>SP090000 | '<br>SP050000 | ý<br>LY110000 | ¸<br>SD410000 | Ý<br>LY120000 | ¨<br>SD170000 | ò<br>LO130000 | ù<br>LU130000 | Ò<br>LO140000 | Ù<br>LU140000 |
| -E | +<br>SA010000 | ;<br>SP140000 | ><br>SA050000 | =<br>SA040000 | þ<br>LT630000 | Æ<br>LA520000 | Þ<br>LT640000 | ´<br>SD110000 | ó<br>LO110000 | ú<br>LU110000 | Ó<br>LO120000 | Ú<br>LU120000 |
| -F | !<br>SP020000 | ^<br>SD150000 | ?<br>SP150000 | "<br>SP040000 | ±<br>SA020000 | ]<br>SM080000 | ®<br>SM530000 | ×<br>SA070000 | õ<br>LO190000 | ÿ<br>LY170000 | Õ<br>LO200000 | (EO) |

**Code Page 00278**

*Figure 116. Code Page IBM-278*

## Code Page IBM-280

| HEX DIGITS<br>1ST →<br>2ND ↓ | 4- | 5- | 6- | 7- | 8- | 9- | A- | B- | C- | D- | E- | F- |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| -0 | (SP)<br>SP010000 | &<br>SM030000 | -<br>SP100000 | ø<br>LO610000 | Ø<br>LO620000 | [<br>SM060000 | µ<br>SM170000 | ¢<br>SC040000 | à<br>LA130000 | è<br>LE130000 | ç<br>LC410000 | 0<br>ND100000 |
| -1 | (RSP)<br>SP300000 | ]<br>SM080000 | /<br>SP120000 | É<br>LE120000 | a<br>LA010000 | j<br>LJ010000 | ì<br>LI130000 | #<br>SM010000 | A<br>LA020000 | J<br>LJ020000 | ÷<br>SA060000 | 1<br>ND010000 |
| -2 | â<br>LA150000 | ê<br>LE150000 | Â<br>LA160000 | Ê<br>LE160000 | b<br>LB010000 | k<br>LK010000 | s<br>LS010000 | ¥<br>SC050000 | B<br>LB020000 | K<br>LK020000 | S<br>LS020000 | 2<br>ND020000 |
| -3 | ä<br>LA170000 | ë<br>LE170000 | Ä<br>LA180000 | Ë<br>LE180000 | c<br>LC010000 | l<br>LL010000 | t<br>LT010000 | ·<br>SD630000 | C<br>LC020000 | L<br>LL020000 | T<br>LT020000 | 3<br>ND030000 |
| -4 | {<br>SM110000 | }<br>SM140000 | À<br>LA140000 | È<br>LE140000 | d<br>LD010000 | m<br>LM010000 | u<br>LU010000 | ©<br>SM520000 | D<br>LD020000 | M<br>LM020000 | U<br>LU020000 | 4<br>ND040000 |
| -5 | á<br>LA110000 | í<br>LI110000 | Á<br>LA120000 | Í<br>LI120000 | e<br>LE010000 | n<br>LN010000 | v<br>LV010000 | @<br>SM050000 | E<br>LE020000 | N<br>LN020000 | V<br>LV020000 | 5<br>ND050000 |
| -6 | ã<br>LA190000 | î<br>LI150000 | Ã<br>LA200000 | Î<br>LI160000 | f<br>LF010000 | o<br>LO010000 | w<br>LW010000 | ¶<br>SM250000 | F<br>LF020000 | O<br>LO020000 | W<br>LW020000 | 6<br>ND060000 |
| -7 | å<br>LA270000 | ï<br>LI170000 | Å<br>LA280000 | Ï<br>LI180000 | g<br>LG010000 | p<br>LP010000 | x<br>LX010000 | ¼<br>NF040000 | G<br>LG020000 | P<br>LP020000 | X<br>LX020000 | 7<br>ND070000 |
| -8 | \<br>SM070000 | ~<br>SD190000 | Ç<br>LC420000 | Ì<br>LI140000 | h<br>LH010000 | q<br>LQ010000 | y<br>LY010000 | ½<br>NF010000 | H<br>LH020000 | Q<br>LQ020000 | Y<br>LY020000 | 8<br>ND080000 |
| -9 | ñ<br>LN190000 | ß<br>LS610000 | Ñ<br>LN200000 | ù<br>LU130000 | i<br>LI010000 | r<br>LR010000 | z<br>LZ010000 | ¾<br>NF050000 | I<br>LI020000 | R<br>LR020000 | Z<br>LZ020000 | 9<br>ND090000 |
| -A | °<br>SM190000 | é<br>LE110000 | ò<br>LO130000 | :<br>SP130000 | «<br>SP170000 | ª<br>SM210000 | ¡<br>SP030000 | ¬<br>SM660000 | (SHY)<br>SP320000 | ¹<br>ND011000 | ²<br>ND021000 | ³<br>ND031000 |
| -B | .<br>SP110000 | $<br>SC030000 | ,<br>SP080000 | £<br>SC020000 | »<br>SP180000 | º<br>SM200000 | ¿<br>SP160000 | \|<br>SM130000 | ô<br>LO150000 | û<br>LU150000 | Ô<br>LO160000 | Û<br>LU160000 |
| -C | <<br>SA030000 | *<br>SM040000 | %<br>SM020000 | §<br>SM240000 | ð<br>LD630000 | æ<br>LA510000 | Đ<br>LD620000 | ¯<br>SM150000 | ö<br>LO170000 | ü<br>LU170000 | Ö<br>LO180000 | Ü<br>LU180000 |
| -D | (<br>SP060000 | )<br>SP070000 | _<br>SP090000 | '<br>SP050000 | ý<br>LY110000 | ¸<br>SD410000 | Ý<br>LY120000 | ¨<br>SD170000 | ¦<br>SM650000 | `<br>SD130000 | Ò<br>LO140000 | Ù<br>LU140000 |
| -E | +<br>SA010000 | ;<br>SP140000 | ><br>SA050000 | =<br>SA040000 | þ<br>LT630000 | Æ<br>LA520000 | Þ<br>LT640000 | ´<br>SD110000 | ó<br>LO110000 | ú<br>LU110000 | Ó<br>LO120000 | Ú<br>LU120000 |
| -F | !<br>SP020000 | ^<br>SD150000 | ?<br>SP150000 | "<br>SP040000 | ±<br>SA020000 | ¤<br>SC010000 | ®<br>SM530000 | ×<br>SA070000 | õ<br>LO190000 | ÿ<br>LY170000 | Õ<br>LO200000 | (EO) |

**Code Page 00280**

*Figure 117. Code Page IBM-280*

## Code Page IBM-284

| HEX DIGITS 1ST → 2ND ↓ | 4- | 5- | 6- | 7- | 8- | 9- | A- | B- | C- | D- | E- | F- |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| -0 | (SP) SP010000 | & SM030000 | - SP100000 | ø LO610000 | Ø LO620000 | ° SM190000 | µ SM170000 | ¢ SC040000 | { SM110000 | } SM140000 | \ SM070000 | 0 ND100000 |
| -1 | (RSP) SP300000 | é LE110000 | / SP120000 | É LE120000 | a LA010000 | j LJ010000 | ¨ SD170000 | £ SC020000 | A LA020000 | J LJ020000 | ÷ SA060000 | 1 ND010000 |
| -2 | â LA150000 | ê LE150000 | Â LA160000 | Ê LE160000 | b LB010000 | k LK010000 | s LS010000 | ¥ SC050000 | B LB020000 | K LK020000 | S LS020000 | 2 ND020000 |
| -3 | ä LA170000 | ë LE170000 | Ä LA180000 | Ë LE180000 | c LC010000 | l LL010000 | t LT010000 | · SD630000 | C LC020000 | L LL020000 | T LT020000 | 3 ND030000 |
| -4 | à LA130000 | è LE130000 | À LA140000 | È LE140000 | d LD010000 | m LM010000 | u LU010000 | © SM520000 | D LD020000 | M LM020000 | U LU020000 | 4 ND040000 |
| -5 | á LA110000 | í LI110000 | Á LA120000 | Í LI120000 | e LE010000 | n LN010000 | v LV010000 | § SM240000 | E LE020000 | N LN020000 | V LV020000 | 5 ND050000 |
| -6 | ã LA190000 | î LI150000 | Ã LA200000 | Î LI160000 | f LF010000 | o LO010000 | w LW010000 | ¶ SM250000 | F LF020000 | O LO020000 | W LW020000 | 6 ND060000 |
| -7 | å LA270000 | ï LI170000 | Å LA280000 | Ï LI180000 | g LG010000 | p LP010000 | x LX010000 | ¼ NF040000 | G LG020000 | P LP020000 | X LX020000 | 7 ND070000 |
| -8 | ç LC410000 | ì LI130000 | Ç LC420000 | Ì LI140000 | h LH010000 | q LQ010000 | y LY010000 | ½ NF010000 | H LH020000 | Q LQ020000 | Y LY020000 | 8 ND080000 |
| -9 | ¦ SM650000 | ß LS610000 | # SM010000 | ` SD130000 | i LI010000 | r LR010000 | z LZ010000 | ¾ NF050000 | I LI020000 | R LR020000 | Z LZ020000 | 9 ND090000 |
| -A | [ SM060000 | ] SM080000 | ñ LN190000 | : SP130000 | « SP170000 | ª SM210000 | ¡ SP030000 | ^ SD150000 | (SHY) SP320000 | ¹ ND011000 | ² ND021000 | ³ ND031000 |
| -B | . SP110000 | $ SC030000 | , SP080000 | Ñ LN200000 | » SP180000 | º SM200000 | ¿ SP160000 | ! SP020000 | ô LO150000 | û LU150000 | Ô LO160000 | Û LU160000 |
| -C | < SA030000 | * SM040000 | % SM020000 | @ SM050000 | ð LD630000 | æ LA510000 | Đ LD620000 | ¯ SM150000 | ö LO170000 | ü LU170000 | Ö LO180000 | Ü LU180000 |
| -D | ( SP060000 | ) SP070000 | _ SP090000 | ' SP050000 | ý LY110000 | ¸ SD410000 | Ý LY120000 | ~ SD190000 | ò LO130000 | ù LU130000 | Ò LO140000 | Ù LU140000 |
| -E | + SA010000 | ; SP140000 | > SA050000 | = SA040000 | þ LT630000 | Æ LA520000 | Þ LT640000 | ´ SD110000 | ó LO110000 | ú LU110000 | Ó LO120000 | Ú LU120000 |
| -F | \| SM130000 | ¬ SM660000 | ? SP150000 | " SP040000 | ± SA020000 | ¤ SC010000 | ® SM530000 | × SA070000 | õ LO190000 | ÿ LY170000 | Õ LO200000 | (EO) |

**Code Page 00284**

*Figure 118. Code Page IBM-284*

## Code Page IBM-285

| HEX DIGITS 1ST → 2ND ↓ | 4- | 5- | 6- | 7- | 8- | 9- | A- | B- | C- | D- | E- | F- |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| -0 | (SP) SP010000 | & SM030000 | - SP100000 | ø LO610000 | Ø LO620000 | ° SM190000 | µ SM170000 | ¢ SC040000 | { SM110000 | } SM140000 | \ SM070000 | 0 ND100000 |
| -1 | (RSP) SP300000 | é LE110000 | / SP120000 | É LE120000 | a LA010000 | j LJ010000 | ‾ SM150000 | [ SM060000 | A LA020000 | J LJ020000 | ÷ SA060000 | 1 ND010000 |
| -2 | â LA150000 | ê LE150000 | Â LA160000 | Ê LE160000 | b LB010000 | k LK010000 | s LS010000 | ¥ SC050000 | B LB020000 | K LK020000 | S LS020000 | 2 ND020000 |
| -3 | ä LA170000 | ë LE170000 | Ä LA180000 | Ë LE180000 | c LC010000 | l LL010000 | t LT010000 | · SD630000 | C LC020000 | L LL020000 | T LT020000 | 3 ND030000 |
| -4 | à LA130000 | è LE130000 | À LA140000 | È LE140000 | d LD010000 | m LM010000 | u LU010000 | © SM520000 | D LD020000 | M LM020000 | U LU020000 | 4 ND040000 |
| -5 | á LA110000 | í LI110000 | Á LA120000 | Í LI120000 | e LE010000 | n LN010000 | v LV010000 | § SM240000 | E LE020000 | N LN020000 | V LV020000 | 5 ND050000 |
| -6 | ã LA190000 | î LI150000 | Ã LA200000 | Î LI160000 | f LF010000 | o LO010000 | w LW010000 | ¶ SM250000 | F LF020000 | O LO020000 | W LW020000 | 6 ND060000 |
| -7 | å LA270000 | ï LI170000 | Å LA280000 | Ï LI180000 | g LG010000 | p LP010000 | x LX010000 | ¼ NF040000 | G LG020000 | P LP020000 | X LX020000 | 7 ND070000 |
| -8 | ç LC410000 | ì LI130000 | Ç LC420000 | Ì LI140000 | h LH010000 | q LQ010000 | y LY010000 | ½ NF010000 | H LH020000 | Q LQ020000 | Y LY020000 | 8 ND080000 |
| -9 | ñ LN190000 | ß LS610000 | Ñ LN200000 | ` SD130000 | i LI010000 | r LR010000 | z LZ010000 | ¾ NF050000 | I LI020000 | R LR020000 | Z LZ020000 | 9 ND090000 |
| -A | $ SC030000 | ! SP020000 | ¦ SM650000 | : SP130000 | « SP170000 | ª SM210000 | ¡ SP030000 | ^ SD150000 | (SHY) SP320000 | ¹ ND011000 | ² ND021000 | ³ ND031000 |
| -B | . SP110000 | £ SC020000 | , SP080000 | # SM010000 | » SP180000 | º SM200000 | ¿ SP160000 | ] SM080000 | ô LO150000 | û LU150000 | Ô LO160000 | Û LU160000 |
| -C | < SA030000 | * SM040000 | % SM020000 | @ SM050000 | ð LD630000 | æ LA510000 | Đ LD620000 | ~ SD190000 | ö LO170000 | ü LU170000 | Ö LO180000 | Ü LU180000 |
| -D | ( SP060000 | ) SP070000 | _ SP090000 | ' SP050000 | ý LY110000 | ¸ SD410000 | Ý LY120000 | ¨ SD170000 | ò LO130000 | ù LU130000 | Ò LO140000 | Ù LU140000 |
| -E | + SA010000 | ; SP140000 | > SA050000 | = SA040000 | þ LT630000 | Æ LA520000 | Þ LT640000 | ´ SD110000 | ó LO110000 | ú LU110000 | Ó LO120000 | Ú LU120000 |
| -F | \| SM130000 | ¬ SM660000 | ? SP150000 | " SP040000 | ± SA020000 | ¤ SC010000 | ® SM530000 | × SA070000 | õ LO190000 | ÿ LY170000 | Õ LO200000 | (EO) |

**Code Page 00285**

*Figure 119. Code Page IBM-285*

## Code Page IBM-297

| HEX DIGITS 1ST → 2ND ↓ | 4- | 5- | 6- | 7- | 8- | 9- | A- | B- | C- | D- | E- | F- |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| -0 | (SP)<br>SP010000 | &<br>SM030000 | -<br>SP100000 | ø<br>LO610000 | Ø<br>LO620000 | [<br>SM060000 | `<br>SD130000 | ¢<br>SC040000 | é<br>LE110000 | è<br>LE130000 | ç<br>LC410000 | 0<br>ND100000 |
| -1 | (RSP)<br>SP300000 | {<br>SM110000 | /<br>SP120000 | É<br>LE120000 | a<br>LA010000 | j<br>LJ010000 | ¨<br>SD170000 | #<br>SM010000 | A<br>LA020000 | J<br>LJ020000 | ÷<br>SA060000 | 1<br>ND010000 |
| -2 | â<br>LA150000 | ê<br>LE150000 | Â<br>LA160000 | Ê<br>LE160000 | b<br>LB010000 | k<br>LK010000 | s<br>LS010000 | ¥<br>SC050000 | B<br>LB020000 | K<br>LK020000 | S<br>LS020000 | 2<br>ND020000 |
| -3 | ä<br>LA170000 | ë<br>LE170000 | Ä<br>LA180000 | Ë<br>LE180000 | c<br>LC010000 | l<br>LL010000 | t<br>LT010000 | ·<br>SD630000 | C<br>LC020000 | L<br>LL020000 | T<br>LT020000 | 3<br>ND030000 |
| -4 | @<br>SM050000 | }<br>SM140000 | À<br>LA140000 | È<br>LE140000 | d<br>LD010000 | m<br>LM010000 | u<br>LU010000 | ©<br>SM520000 | D<br>LD020000 | M<br>LM020000 | U<br>LU020000 | 4<br>ND040000 |
| -5 | á<br>LA110000 | í<br>LI110000 | Á<br>LA120000 | Í<br>LI120000 | e<br>LE010000 | n<br>LN010000 | v<br>LV010000 | ]<br>SM080000 | E<br>LE020000 | N<br>LN020000 | V<br>LV020000 | 5<br>ND050000 |
| -6 | ã<br>LA190000 | î<br>LI150000 | Ã<br>LA200000 | Î<br>LI160000 | f<br>LF010000 | o<br>LO010000 | w<br>LW010000 | ¶<br>SM250000 | F<br>LF020000 | O<br>LO020000 | W<br>LW020000 | 6<br>ND060000 |
| -7 | å<br>LA270000 | ï<br>LI170000 | Å<br>LA280000 | Ï<br>LI180000 | g<br>LG010000 | p<br>LP010000 | x<br>LX010000 | ¼<br>NF040000 | G<br>LG020000 | P<br>LP020000 | X<br>LX020000 | 7<br>ND070000 |
| -8 | \<br>SM070000 | ì<br>LI130000 | Ç<br>LC420000 | Ì<br>LI140000 | h<br>LH010000 | q<br>LQ010000 | y<br>LY010000 | ½<br>NF010000 | H<br>LH020000 | Q<br>LQ020000 | Y<br>LY020000 | 8<br>ND080000 |
| -9 | ñ<br>LN190000 | ß<br>LS610000 | Ñ<br>LN200000 | µ<br>SM170000 | i<br>LI010000 | r<br>LR010000 | z<br>LZ010000 | ¾<br>NF050000 | I<br>LI020000 | R<br>LR020000 | Z<br>LZ020000 | 9<br>ND090000 |
| -A | °<br>SM190000 | §<br>SM240000 | ù<br>LU130000 | :<br>SP130000 | «<br>SP170000 | ª<br>SM210000 | ¡<br>SP030000 | ¬<br>SM660000 | (SHY)<br>SP320000 | ¹<br>ND011000 | ²<br>ND021000 | ³<br>ND031000 |
| -B | .<br>SP110000 | $<br>SC030000 | ,<br>SP080000 | £<br>SC020000 | »<br>SP180000 | º<br>SM200000 | ¿<br>SP160000 | \|<br>SM130000 | ô<br>LO150000 | û<br>LU150000 | Ô<br>LO160000 | Û<br>LU160000 |
| -C | <<br>SA030000 | *<br>SM040000 | %<br>SM020000 | à<br>LA130000 | ð<br>LD630000 | æ<br>LA510000 | Đ<br>LD620000 | ¯<br>SM150000 | ö<br>LO170000 | ü<br>LU170000 | Ö<br>LO180000 | Ü<br>LU180000 |
| -D | (<br>SP060000 | )<br>SP070000 | _<br>SP090000 | '<br>SP050000 | ý<br>LY110000 | ¸<br>SD410000 | Ý<br>LY120000 | ~<br>SD190000 | ò<br>LO130000 | ¦<br>SM650000 | Ò<br>LO140000 | Ù<br>LU140000 |
| -E | +<br>SA010000 | ;<br>SP140000 | ><br>SA050000 | =<br>SA040000 | þ<br>LT630000 | Æ<br>LA520000 | Þ<br>LT640000 | ´<br>SD110000 | ó<br>LO110000 | ú<br>LU110000 | Ó<br>LO120000 | Ú<br>LU120000 |
| -F | !<br>SP020000 | ^<br>SD150000 | ?<br>SP150000 | "<br>SP040000 | ±<br>SA020000 | ¤<br>SC010000 | ®<br>SM530000 | ×<br>SA070000 | õ<br>LO190000 | ÿ<br>LY170000 | Õ<br>LO200000 | (EO) |

**Code Page 00297**

*Figure 120. Code Page IBM-297*

## Code Page IBM-500

| HEX DIGITS 1ST → 2ND ↓ | 4- | 5- | 6- | 7- | 8- | 9- | A- | B- | C- | D- | E- | F- |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| -0 | (SP) SP010000 | & SM030000 | - SP100000 | ø LO610000 | Ø LO620000 | ° SM190000 | µ SM170000 | ¢ SC040000 | { SM110000 | } SM140000 | \ SM070000 | 0 ND100000 |
| -1 | (RSP) SP300000 | é LE110000 | / SP120000 | É LE120000 | a LA010000 | j LJ010000 | ~ SD190000 | £ SC020000 | A LA020000 | J LJ020000 | ÷ SA060000 | 1 ND010000 |
| -2 | â LA150000 | ê LE150000 | Â LA160000 | Ê LE160000 | b LB010000 | k LK010000 | s LS010000 | ¥ SC050000 | B LB020000 | K LK020000 | S LS020000 | 2 ND020000 |
| -3 | ä LA170000 | ë LE170000 | Ä LA180000 | Ë LE180000 | c LC010000 | l LL010000 | t LT010000 | · SD630000 | C LC020000 | L LL020000 | T LT020000 | 3 ND030000 |
| -4 | à LA130000 | è LE130000 | À LA140000 | È LE140000 | d LD010000 | m LM010000 | u LU010000 | © SM520000 | D LD020000 | M LM020000 | U LU020000 | 4 ND040000 |
| -5 | á LA110000 | í LI110000 | Á LA120000 | Í LI120000 | e LE010000 | n LN010000 | v LV010000 | § SM240000 | E LE020000 | N LN020000 | V LV020000 | 5 ND050000 |
| -6 | ã LA190000 | î LI150000 | Ã LA200000 | Î LI160000 | f LF010000 | o LO010000 | w LW010000 | ¶ SM250000 | F LF020000 | O LO020000 | W LW020000 | 6 ND060000 |
| -7 | å LA270000 | ï LI170000 | Å LA280000 | Ï LI180000 | g LG010000 | p LP010000 | x LX010000 | ¼ NF040000 | G LG020000 | P LP020000 | X LX020000 | 7 ND070000 |
| -8 | ç LC410000 | ì LI130000 | Ç LC420000 | Ì LI140000 | h LH010000 | q LQ010000 | y LY010000 | ½ NF010000 | H LH020000 | Q LQ020000 | Y LY020000 | 8 ND080000 |
| -9 | ñ LN190000 | ß LS610000 | Ñ LN200000 | ` SD130000 | i LI010000 | r LR010000 | z LZ010000 | ¾ NF050000 | I LI020000 | R LR020000 | Z LZ020000 | 9 ND090000 |
| -A | [ SM060000 | ] SM080000 | ¦ SM650000 | : SP130000 | « SP170000 | ª SM210000 | ¡ SP030000 | ¬ SM660000 | (SHY) SP320000 | ¹ ND011000 | ² ND021000 | ³ ND031000 |
| -B | . SP110000 | $ SC030000 | , SP080000 | # SM010000 | » SP180000 | º SM200000 | ¿ SP160000 | \| SM130000 | ô LO150000 | û LU150000 | Ô LO160000 | Û LU160000 |
| -C | < SA030000 | * SM040000 | % SM020000 | @ SM050000 | ð LD630000 | æ LA510000 | Đ LD620000 | ¯ SM150000 | ö LO170000 | ü LU170000 | Ö LO180000 | Ü LU180000 |
| -D | ( SP060000 | ) SP070000 | _ SP090000 | ' SP050000 | ý LY110000 | ¸ SD410000 | Ý LY120000 | ¨ SD170000 | ò LO130000 | ù LU130000 | Ò LO140000 | Ù LU140000 |
| -E | + SA010000 | ; SP140000 | > SA050000 | = SA040000 | þ LT630000 | Æ LA520000 | Þ LT640000 | ´ SD110000 | ó LO110000 | ú LU110000 | Ó LO120000 | Ú LU120000 |
| -F | ! SP020000 | ^ SD150000 | ? SP150000 | " SP040000 | ± SA020000 | ¤ SC010000 | ® SM530000 | × SA070000 | õ LO190000 | ÿ LY170000 | Õ LO200000 | (EO) |

**Code Page 00500**

*Figure 121. Code Page IBM-500*

## Code Page IBM-871

| HEX DIGITS 1ST → 2ND ↓ | 4- | 5- | 6- | 7- | 8- | 9- | A- | B- | C- | D- | E- | F- |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| -0 | (SP) SP010000 | & SM030000 | - SP100000 | ø LO610000 | Ø LO620000 | ° SM190000 | µ SM170000 | ¢ SC040000 | þ LT630000 | æ LA510000 | ´ SD110000 | 0 ND100000 |
| -1 | (RSP) SP300000 | é LE110000 | / SP120000 | É LE120000 | a LA010000 | j LJ010000 | ö LO170000 | £ SC020000 | A LA020000 | J LJ020000 | ÷ SA060000 | 1 ND010000 |
| -2 | â LA150000 | ê LE150000 | Â LA160000 | Ê LE160000 | b LB010000 | k LK010000 | s LS010000 | ¥ SC050000 | B LB020000 | K LK020000 | S LS020000 | 2 ND020000 |
| -3 | ä LA170000 | ë LE170000 | Ä LA180000 | Ë LE180000 | c LC010000 | l LL010000 | t LT010000 | · SD630000 | C LC020000 | L LL020000 | T LT020000 | 3 ND030000 |
| -4 | à LA130000 | è LE130000 | À LA140000 | È LE140000 | d LD010000 | m LM010000 | u LU010000 | © SM520000 | D LD020000 | M LM020000 | U LU020000 | 4 ND040000 |
| -5 | á LA110000 | í LI110000 | Á LA120000 | Í LI120000 | e LE010000 | n LN010000 | v LV010000 | § SM240000 | E LE020000 | N LN020000 | V LV020000 | 5 ND050000 |
| -6 | ã LA190000 | î LI150000 | Ã LA200000 | Î LI160000 | f LF010000 | o LO010000 | w LW010000 | ¶ SM250000 | F LF020000 | O LO020000 | W LW020000 | 6 ND060000 |
| -7 | å LA270000 | ï LI170000 | Å LA280000 | Ï LI180000 | g LG010000 | p LP010000 | x LX010000 | ¼ NF040000 | G LG020000 | P LP020000 | X LX020000 | 7 ND070000 |
| -8 | ç LC410000 | ì LI130000 | Ç LC420000 | Ì LI140000 | h LH010000 | q LQ010000 | y LY010000 | ½ NF010000 | H LH020000 | Q LQ020000 | Y LY020000 | 8 ND080000 |
| -9 | ñ LN190000 | ß LS610000 | Ñ LN200000 | ð LD630000 | i LI010000 | r LR010000 | z LZ010000 | ¾ NF050000 | I LI020000 | R LR020000 | Z LZ020000 | 9 ND090000 |
| -A | Þ LT640000 | Æ LA520000 | ¦ SM650000 | : SP130000 | « SP170000 | ª SM210000 | ¡ SP030000 | ¬ SM660000 | (SHY) SP320000 | ¹ ND011000 | ² ND021000 | ³ ND031000 |
| -B | . SP110000 | $ SC030000 | , SP080000 | # SM010000 | » SP180000 | º SM200000 | ¿ SP160000 | \| SM130000 | ô LO150000 | û LU150000 | Ô LO160000 | Û LU160000 |
| -C | < SA030000 | * SM040000 | % SM020000 | Đ LD620000 | ` SD130000 | } SM140000 | @ SM050000 | ¯ SM150000 | ~ SD190000 | ü LU170000 | ^ SD150000 | Ü LU180000 |
| -D | ( SP060000 | ) SP070000 | _ SP090000 | ' SP050000 | ý LY110000 | ¸ SD410000 | Ý LY120000 | ¨ SD170000 | ò LO130000 | ù LU130000 | Ò LO140000 | Ù LU140000 |
| -E | + SA010000 | ; SP140000 | > SA050000 | = SA040000 | { SM110000 | ] SM080000 | [ SM060000 | \ SM070000 | ó LO110000 | ú LU110000 | Ó LO120000 | Ú LU120000 |
| -F | ! SP020000 | Ö LO180000 | ? SP150000 | " SP040000 | ± SA020000 | ¤ SC010000 | ® SM530000 | × SA070000 | õ LO190000 | ÿ LY170000 | Õ LO200000 | (EO) |

**Code Page 00871**

*Figure 122. Code Page IBM-871*

# Appendix C. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:** INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

## Trademarks

# Glossary

This glossary defines technical terms and abbreviations used in z/OS DCE documentation. If you do not find the term you are looking for, refer to the index of the appropriate z/OS DCE manual or view the *IBM Glossary of Computing Terms*, located at:

**http://www.ibm.com/ibm/terminology**

This glossary includes terms and definitions from:

- *IBM Dictionary of Computing*, SC20-1699.
- *Information Technology—Portable Operating System Interface (POSIX),* from the POSIX series of standards for applications and user interfaces to open systems, copyrighted by the Institute of Electrical and Electronics Engineers (IEEE).
- *American National Standard Dictionary for Information Systems,* ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 11 West 42nd Street, New York, New York 10036. Definitions are identified by the symbol (A) after the definition.
- *Information Technology Vocabulary,* developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1.SC1).
- *CCITT Sixth Plenary Assembly Orange Book, Terms and Definitions* and working documents published by the International Telecommunication Union, Geneva, 1978.
- Open Software Foundation (OSF).

The following abbreviations indicate terms that are related to a particular DCE service:

| | |
|---|---|
| **CDS** | Cell Directory Service |
| **CICS/ESA®** | Customer Information Control System/ESA |
| **DTS** | Distributed Time Service |
| **GDS** | Global Directory Service |
| **IMS/ESA®** | Information Management System/ESA |
| **RPC** | Remote Procedure Call |
| **Security** | Security Service |
| **Threads** | Threads Service |
| **XDS** | X/Open Directory Services |
| **XOM** | X/Open OSI-Abstract-Data Manipulation |

## A

**absolute time**. A point on a time scale.

**access control list (ACL)**. (1) GDS: Specifies the users with their access rights to an object. (2) Security: Data that controls access to a protected object. An ACL specifies the privilege attributes needed to access the object and the permissions that may be granted, to the protected object, to principals that possess such privilege attributes.

**access control list facility**. A Security Service feature that checks a principal's access to an object. This facility determines access rights by comparing the principal's privileges to entries in an access control list (ACL) of an object.

**access right**. Synonym for *permission*.

**accessible**. Pertaining to an object whose client possesses a valid designator or handle.

**account**. Data in the Registry database that allows a principal to log in. An account is a registry object that relates to a principal.

**ACF**. Attribute configuration file.

**ACL**. Access control list.

**active context handle**. RPC: A context handle in RPC applications that the RPC has set to a non-null value and passed back to the calling program. The calling program supplies the active context handle in any future calls to procedures that share the same client context. See *client context* and *context handle*.

**address**. An unambiguous name, label, or number that identifies the location of a particular entity or service. See *presentation address*.

**address family**. A set of related communications protocols that use a common addressing mechanism to identify end-points; for example, the U.S. Department of Defense Internet Protocols. Synonymous with *protocol family*.

**alias**. Synonym for *alias name*.

**alias name**. (1) GDS: A name for a directory object that consists of one or more alias entries in the directory information tree (DIT). (2) Security: An optional alternate for a principal's primary name.

Synonymous with *alias*. The alias shares the same UUID with the primary name.

**aliasing**. RPC: Pertaining to the pointing of two pointers of the same operation at the same storage.

**API**. Application program interface.

**application program interface (API)**. A functional interface supplied by the operating system or by a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed program.

**application thread**. A thread of execution created and managed by application code. See *client application thread*, *local application thread*, *RPC thread*, and *server application thread*.

**architecture**. (1) The organizational structure of a computer system, including the interrelationships among its hardware and software. (2) The logical structure and operating principles of a computer network. The operating principles of a network include those of services, functions, and protocols.

**association (connection-oriented)**. A connection between a client and a server.

**asynchronous**. Without a regular time relationship; unexpected or unpredictable with respect to the running of program instructions.

**at-most-once semantics**. RPC: A characteristic of a procedure that restricts the procedure to being run once, partially, or not at all. See *broadcast semantics*, *idempotent semantics*, and *maybe semantics*.

**attribute**. (1) RPC: An Interface Definition Language (IDL) or attribute configuration file (ACF) that conveys information about an interface, type, field, parameter, or operation. (2) DTS: A qualifier used with DTS commands. DTS has four attribute categories: characteristics, counters, identifiers, and status.
(3) XDS: Information of a particular type concerning an object and appearing in an entry that describes the object in the directory information base (DIB). It denotes the attribute's type and a sequence of one or more attribute values, each accompanied by an integer denoting the value's syntax.

**attribute configuration file (ACF)**. RPC: An optional companion to an interface definition file that changes how the Interface Definition Language (IDL) compiler locally interprets the interface definition. See also *interface definition* and *Interface Definition Language*.

**Attribute Configuration Language**. RPC: A high-level declarative language that provides syntax for attribute configuration files. See *attribute configuration file*.

**attribute syntax**. GDS: A definition of the set of values that an attribute may assume. Attribute syntax includes the data type, in ASN.1, and usually one or more matching rules by which values may be compared.

**attribute type**. (1) XDS: The component of an attribute that indicates the type of information given by that attribute. Because it is an object identifier, it is unique among other attribute types. (2) XOM: Any of various categories into which the client dynamically groups values on the basis of their semantics. It is an integer unique only within the package.

**attribute value**. XDS, XOM: A particular instance of the type of information indicated by an attribute type.

**authentication**. In computer security, a method used to verify the identity of a principal.

**authentication level**. Synonym for *protection level*.

**authentication protocol**. A formal procedure for verifying a principal's network identity. Kerberos is an instance of a shared-secret authentication protocol.

**Authentication Service**. One of three services provided by the Security Service: it verifies principals according to a specified authentication protocol. The other Security services are the Privilege Service and the Registry Service.

**authentication surrogate**. Security: A type of principal entry in a cell's Registry database that represents a foreign cell. This principal shares a secret key with a corresponding entry in the foreign cell's Registry. The Authentication Services of the two cells use the secret key to exchange data about principals without either Authentication Service having to share its private key with the other.

**authorization**. (1) The determination of a principal's permissions with respect to a protected object. (2) The approval of a permission sought by a principal with respect to a protected object.

**authorization protocol**. A formal procedure for establishing the authorization of principals with respect to protected objects. Authorization protocols supported by the Security Service include DCE authorization and name-based authorization.

**authorization service**. RPC: An implementation of an authorization protocol.

**automatic binding method**. RPC: A method of managing the binding for a remote procedure call. It completely hides binding management from client application code. If the client makes a series of remote procedure calls, the stub passes the same binding handle with each call. See *binding handle*, *explicit binding method*, and *implicit binding method*.

## B

**big endian**. An attribute of data representation that reflects how multi-octet data is stored. In big endian representation, the lowest addressed octet of a multi-octet data item is the most significant. See *little endian*.

**binary timestamp**. An opaque 128-bit (16-octet) structure that represents a DTS time value.

**binding**. RPC: A relationship between a client and a server involved in a remote procedure call.

**binding handle**. RPC: A reference to a binding. See *binding information*.

**binding information**. RPC: Information about one or more potential bindings, including an RPC protocol sequence, a network address, an endpoint, at least one transfer syntax, and an RPC protocol version number. See *binding*. See also *endpoint*, *network address*, *RPC protocol*, *RPC protocol sequence*, and *transfer syntax*.

**broadcast**. A notification sent to all members within an arbitrary grouping such as nodes in a network or threads in a process. See also *signal*.

**broadcast semantics**. RPC: A form of idempotent semantics that indicates that the operation is always broadcast to all host systems on the local network, rather than delivered to a specific system. An operation with broadcast semantics is implicitly idempotent. Broadcast semantics are supported only by connectionless protocols. See *at-most-once semantics*, *idempotent semantics*, and *maybe semantics*.

**Browser**. CDS: A Motif-based program that lets users view the contents and structure of a cell name space.

## C

**C interface**. The interface that is defined at a level that depends on the variant of C standardized by ANSI.

**cache**. (1) CDS: The information that a CDS clerk stores locally to optimize name lookups. The cache contains attribute values resulting from previous lookups, as well as information about other clearinghouses and namespaces. (2) Security: Contains the credentials of a principal after the DCE login. (3) GDS: See *DUA cache*.

**callback**. A role reversal technique used by the server to make a request back to the original client. For example, the server may request state information (such as sequence numbers) needed to provide reliable data transfer or identity information needed for an authenticated RPC call.

**call handle**. An RPC data structure used by the RPC runtime to maintain the state information for an RPC call. A client call handle is maintained by the client, and a corresponding server call handle is maintained by the server.

**call queue**. RPC: A FIFO queue used by an RPC server to hold incoming calls when the server is already running its maximum number of concurrent calls.

**call thread**. RPC: A thread created by an RPC server's runtime to run remote procedures. When engaged by a remote procedure call, a call thread temporarily forms part of the RPC thread of the call. See *application thread* and *RPC thread*.

**cancel**. (1) Threads: A mechanism by which a thread informs either itself or another thread to stop the thread as soon as possible. If a cancel arrives during an important operation, the canceled thread may continue until it can end the thread in a controlled manner.
(2) RPC: A mechanism by which a client thread notifies a server thread (the canceled thread) to end the thread as soon as possible. See also *thread*.

**CCITT**. Consultative Committee on International Telegraphy and Telephone

**CDS**. Cell Directory Service.

**CDS clerk**. The software that provides an interface between client applications and CDS servers.

**CDS control program (CDSCP)**. A command interface that CDS administrators use to control CDS servers and clerks and manage the name space and its contents. See also *manager*.

**CDSCP**. CDS control program.

**cell**. The basic unit of operation in the distributed computing environment. A cell is a group of users, systems, and resources that are grouped around a common purpose and that share common DCE services.

**Cell Directory Service (CDS)**. A DCE component. A distributed replicated database service that stores names and attributes of resources located in a cell. CDS manages a database of information about the resources in a group of machines called a DCE cell.

**cell-relative name**. Synonym for *local name*.

**central processing unit (CPU)**. The part of a computer that includes the circuits that control the interpretation and processing of instructions.

**child process**. A process, created by a parent process, that shares the resources of the parent process to carry out a request. Contrast with *parent process*. See also *fork*.

**class**. A category into which objects are placed on the basis of their purpose and internal structure.

**clerk**. (1) DTS: A software component that synchronizes the clock for its client system by requesting time values from servers, calculating a new time from the values, and supplying the computed time to client applications. (2) CDS: A software component that receives CDS requests from a client application, ascertains an appropriate CDS server to process the requests, and returns the results of the requests to the client application.

**client**. A computer or process that accesses the data, services, or resources of another computer or process on the network. Contrast with *server*.

**client application thread**. RPC: A thread executing client application code that makes one or more remote procedure calls. See *application thread*, *local application thread*, *RPC thread*, and *server application thread*.

**client binding information**. Information about a calling client provided by the client runtime to the server runtime, including the address where the call originated, the RPC protocol used for the call, the requested object UUID, and client authentication information. See *binding information* and *server binding information*.

**client context**. RPC: The state within an RPC server generated by a set of remote procedures and maintained across a series of calls for a particular client. See *context handle*. See also *manager*.

**client stub**. RPC: The surrogate code for an RPC interface that is linked with and called by the client application code. In addition to general operations such as marshalling data, a client stub calls the RPC runtime to perform remote procedure calls and, optionally, to manage bindings. See *server stub*.

**client/server model**. A form of computing where one system, the client, requests something, and another system, the server, responds.

**clock**. The combined hardware interrupt timer and software register that maintains the system time.

**code page**. (1) A table showing codes assigned to character sets. (2) An assignment of graphic characters and control function meanings to all code points. (3) Arrays of code points representing characters that establish numeric order of characters. [OSF] (4) A particular assignment of hexadecimal identifiers to graphic elements. (5) Synonymous with code set. (6) See also *code point, extended character*.

**code set**. Synonym for *code page*.

**collapse**. CDS: To remove the contents of a directory from the display (close it) using the CDS Browser. To collapse an open directory, double-click on its icon. Double-clicking on a closed directory expands it. Contrast with *expand*.

**communications link**. RPC: A network pathway between an RPC client and server that uses a valid combination of transport and network protocols that are available to both the client and server RPC run times.

**compatible server**. RPC: A server that offers the requested RPC interface and RPC object and that is accessible over a valid combination of network and transport protocols. It is supported by both the client and server RPC run times.

**condition variable**. Threads: A synchronization object used in conjunction with a mutex. It allows a thread to suspend running until some condition is true.

**conformant array**. RPC: An array whose size is determined at runtime. A structure containing a conformant array as a field is a conformant structure.

**connectionless protocol**. RPC: A transport protocol such as UDP that does not require a connection to be established prior to data transfer. Contrast with *connection-oriented protocol*.

**connection-oriented protocol**. RPC: An RPC protocol that runs over a connection-based transport protocol. It is a connection-based, reliable, virtual-circuit transport protocol, such as TCP. Contrast with *connectionless protocol*.

**Consultative Committee on International Telegraphy and Telephone (CCITT)**. A United Nations Specialized Standards group whose membership includes common carriers concerned with devising and proposing recommendations for international telecommunications representing alphabets, graphics, control information, and other fundamental information interchange issues.

**context handle**. RPC: A reference to state (client context) maintained across remote procedure calls by a server on behalf of a client. See *client context*.

**control access**. CDS: An access right that grants users the ability to change the access control on a name and to perform other powerful management tasks, such as replicate a directory or move a clearinghouse.

**conversation key**. Synonym for *session key*.

**copy**. GDS, XDS: Either a copy of an entry stored in other DSAs through bilateral agreement or a locally and dynamically stored copy of an entry resulting from a request (a cache copy).

**CPU**. central processing unit

**creation timestamp (CTS)**. An attribute of all CDS clearinghouses, directories, soft links, child pointers, and object entries that contains a unique value reflecting the date and time the name was created. The timestamp consists of two parts; a time portion and a portion containing the system identifier of the node on which the name was created. These two parts guarantee uniqueness among timestamps generated on different nodes.

**credentials**. Security: A general term for privilege attribute data that has been certified by a trusted privilege certification authority.

**cross-linking information**. In order for z/OS DCE to provide RACF-DCE interoperability and single sign-on to DCE, DCE provides utilities (see **mvsexpt** and **mvsimpt**) to incorporate into RACF the information that associates a z/OS-RACF user ID with a DCE principal's identifying information and the DCE principal's UUID with the corresponding z/OS-RACF user ID. The information is placed in a RACF DCE segment and the RACF general resource class, DCEUUIDS. This is called **cross-linking information** and is what allows interoperability and single sign-on to work. See also **interoperability** and **single sign-on**.

**customized binding handle**. RPC: A user-defined data structure from which a primitive binding handle can be derived by user-defined routines in application code. See *primitive binding handle*.

## D

**daemon**. (1) A long-lived process that runs unattended to perform continuous or periodic system-wide functions such as network control Some daemons are triggered automatically to perform their task; others operate periodically. An example is the **cron** daemon, which periodically performs the tasks listed in the **crontab** file. Many standard dictionaries accept the spelling *demon*. (2) A DCE server process.

**Data Encryption Standard (DES)**. The National Institute of Standards and Technology (NIST) Data Encryption Standard, adopted by the U.S. government as Federal Information Processing Standard (FIPS) Publication 46, which allows only hardware implementations of the data encryption algorithm.

**data limit**. RPC: A value that specifies which elements of an array are transmitted during a remote procedure call.

**datagram**. RPC: A network data packet that is independent of all other packets and does not guarantee delivery or sequentiality.

**datagram protocol**. RPC: A datagram-based transport protocol, such as User Datagram Protocol (UDP), that runs over a connectionless transport protocol.

**DCE**. Distributed Computing Environment.

**DCEKERN**. The address space that contains the DCE daemons.

**decrypt**. Security: To decipher data.

**default element**. RPC: An optional profile element that contains a nil interface identifier and object UUID and that specifies a default profile. Each profile can contain only one default element. See *default profile*, *profile*, and *profile element*.

**default profile**. RPC: A backup profile referred to by the default element in another profile. The NSI import and lookup operations use the default profile, if present, whenever a search based on the current profile fails to find any useful binding information. See *default element* and *profile*.

**DES**. Data Encryption Standard.

**descriptor**. (1) XOM: The means by which the client and service exchange an attribute value and the integers that denote its representation, type, and syntax. (2) XDS: A defined data structure that is used to represent an OM attribute type and a single value.

**destructor**. A user-supplied routine that is expected to finalize and then deallocate a per-thread context value.

**DFS**. Distributed File Service.

**DIB**. Directory information base.

**directory**. (1) A logical unit for storing entries under one name (the directory name) in a CDS namespace. Each physical instance of a directory is called a replica. (2) A collection of open systems that cooperates to hold a logical database of information about a set of objects in the real world.

**directory information base (DIB)**. GDS: The complete set of information to which the directory provides access, which includes all of the pieces of information that can be read or manipulated using the operations of the directory.

**directory schema**. GDS: The set of rules and constraints concerning directory information tree (DIT) structure, object class definitions, attribute types, and syntaxes that characterize the directory information base (DIB).

**Directory Service**. A DCE component. The Directory Service is a central repository for information about resources in a distributed system. See *Cell Directory Service* and *Global Directory Service*.

**discriminator**. RPC: The data item that determines which union case is currently used.

**distributed computing**. A type of computing that allows computers with different hardware and software to be combined on a network, to function as a single computer, and to share the task of processing application programs.

**Distributed Computing Environment (DCE)**. A comprehensive, integrated set of services that supports the development, use, and maintenance of distributed applications. DCE is independent of the operating system and network; it provides interoperability and portability across heterogeneous platforms.

**Distributed File Service (DFS)**. A DCE component. DFS joins the local file systems of several file server machines making the files equally available to all DFS client machines. DFS allows users to access and share files stored on a file server anywhere in the network, without having to consider the physical location of the file. Files are part of a single, global name space, so that a user can be found anywhere in the network by means of the same name.

**distributed service**. A DCE service that is used mainly by administrators to manage a distributed environment. These services include DTS, Security, and Directory.

**Distributed Time Service (DTS)**. A DCE component. It provides a way to synchronize the times on different hosts in a distributed system.

**DNS**. Domain Name System.

**Domain Name System (DNS)**. A hierarchical scheme for giving meaningful names to hosts in a TCP/IP network.

**domain name**. A unique network name that is associated with a network's unique address.

**DTS**. Distributed Time Service.

**DTS entity**. DTS: The server or clerk software on a system.

**DUA cache**. GDS: The part of the DUA that stores information to optimize name lookups. Each cache contains copies of recently accessed object entries as well as information about DSAs in the directory.

**dynamic endpoint**. RPC: An endpoint that is generated by the RPC runtime for an RPC server when the server registers its protocol sequences. It expires when the server stops running. See *endpoint* and *well-known endpoint*.

## E

**effective permissions**. Security: The permissions granted to a principal as a result of a masking operation.

**element**. RPC: Any of the bits of a bit string, the octets of an octet string, or the octets by means of which the characters of a character string are represented.

**encrypt**. To systematically encode data so that it cannot be read without knowing the coding key.

**encryption key**. A value used to encrypt data so that only possessors of the encryption key can decipher it.

**endian**. An attribute of data representation that reflects how certain multi-octet data is stored in memory. See *big endian* and *little endian*.

**endpoint**. RPC: An address of a specific server instance on a host.

**endpoint map**. RPC: A database local to a node where local RPC servers register binding information associated with their interface identifiers and object identifiers. The endpoint map is maintained by the endpoint map service of the DCE daemon.

**endpoint map service**. RPC: A service that maintains a system's endpoint map for local RPC servers. When an RPC client makes a remote procedure call using a partially bound binding handle, the endpoint map service looks up the endpoint of a compatible local server. See *endpoint map*.

**entity**. (1) CDS: Any manageable element through the CDS namespace. Manageable elements include directories, object entries, servers, replicas, and clerks. The CDS control program (CDSCP) commands are based on directives targeted for specific entities.
(2) DTS: See *DTS entity*.

**entry**. GDS, XDS: The part of the DIB that contains information relating to a single directory object. Each entry consists of directory attributes.

**entry point vector (EPV)**. RPC: A list of addresses for the entry points of a set of remote procedures that starts the operations declared in an interface definition. The addresses are listed in the same order as the corresponding operation declarations.

**ENV**. environment variable

**envelope**. Security: Used to transport authentication data and conversation keys between the security server and principals.

**environment variable (ENV)**. A variable included in the current software environment that is available to any called program that requests it.

**EPV**. Entry point vector.

**exception**. (1) An abnormal condition such as an I/O error encountered in processing a data set or a file. (2) One of five types of errors that can occur during a floating-point exception. These are valid operation, overflow, underflow, division by zero, and inexact results. [OSF] (3) Contrast with *interrupt, signal*.

**executor thread**. See *call thread*.

**expand**. CDS: To display the contents of (open) a directory using the CDS Browser. A directory that is closed can be expanded by double-clicking on its icon. Double-clicking on an expanded directory collapses it. Contrast with *collapse*.

**expiration age**. RPC: The amount of time that a local copy of name service data from a NSI attribute remains unchanged before a request from an RPC application for the attribute requires its updating. See also *NSI attribute*.

**explicit binding method**. RPC: The explicit method of managing the binding for a remote procedure call in which a remote procedure call passes a binding handle as its first parameter. The binding handle is initialized in the application code. See *automatic binding method*, *binding handle*, and *implicit binding method*.

**export**. (1) RPC: To place the server binding information associated with an RPC interface or a list of object UUIDs or both into an entry in a name service database. (2) To provide access information for an RPC interface. Contrast with *unexport*.

## F

**fault**. RPC: An exception condition, occurring on a server, that is transmitted to a client.

**filter**. An assertion about the presence or value of certain attributes of an entry to limit the scope of a search.

**FIFO**. first-in-first-out

**first-in-first-out (FIFO)**. A queueing technique in which the next item to be retrieved is the item that has been in the queue the longest time.

**fixed array**. RPC: The size of the array is defined in the IDL. All of the data in the array is transmitted during a remote procedure call.

**foreign cell**. A cell other than the one to which the local machine belongs. A foreign cell and its binding information are stored in either GDS or the Domain Name System (DNS). The act of contacting a foreign cell is called intercell. Contrast with *local cell*.

**fork**. To create and start a child process. Forking is similar to creating an address space and attaching. It creates a copy of the parent process, including open file descriptors.

**full name**. CDS: The complete specification of a CDS name, including all parent directories in the path from the cell root to the entry being named.

**full pointer**. RPC: A pointer without the restrictions of a reference pointer.

**fully bound binding handle**. RPC: A server binding handle that contains a complete server address including an endpoint. Contrast with *partially bound binding handle*.

## G

**General-Use Programming Interface (GUPI)**. An interface, with few restrictions, for use in customer-written programs. The majority of programming interfaces are general-use programming interfaces, and are appropriate in a wide variety of application programs. A general-use programming interface requires the knowledge of the externals of the interface and perhaps the externals of related programming interfaces. Knowledge of the detailed design or implementation of the software product is not required.

**GDS**. Global Directory Service.

**Global Directory Agent (GDA)**. A DCE component that makes it possible for the local CDS to access names in foreign cells. The GDA provides a connection to foreign cells through either the GDS or the Domain Name System (DNS).

**Global Directory Service (GDS)**. A DCE component. A distributed replicated directory service that provides a global namespace that connects the local DCE cells into one worldwide hierarchy. DCE users can look up a name outside a local cell with GDS.

**global name**. A name that is universally meaningful and usable from anywhere in the DCE naming environment. The prefix /... indicates that a name is global.

**global server**. DTS: A server that provides its clock value to courier servers on other cells, or to DTS entities that have failed to obtain the specified number of servers locally.

**group**. (1) RPC: A name service entry that corresponds to one or more RPC servers that offer common RPC interfaces, RPC objects, or both. A group contains the names of the server entries, other groups, or both that are members of the group. See *NSI group attribute*. (2) Security: Data that associates a named set of principals that can be granted common access rights. See *subject identifier*.

**group member**. (1) RPC: A name service entry whose name occurs in the group. (2) Security: A principal whose name appears in a security group. See *group*.

## H

**handle**. RPC: An opaque reference to information. See *binding handle*, *context handle*, *interface handle*, *name service handle*, and *thread handle*.

**heterogeneous**. Pertaining to a collection of dissimilar host computers such as those from different manufacturers. Contrast with *homogeneous*.

**home cell**. Synonym for *local cell*.

**homogeneous**. Pertaining to a collection of similar host computers such as those of one model or one manufacturer. Contrast with *heterogeneous*.

**host ID**. Synonym for *network address*.

## I

**idempotent semantics**. RPC: A characteristic of a procedure in which running it more than once with identical input always produces the same result, without any undesirable side effects. For example, a procedure that calculates the square root of a number is idempotent. DCE RPC supports maybe and broadcast semantics as special forms of idempotent operations. See *at-most-once semantics*, *broadcast semantics*, and *maybe semantics*.

**IDL**. Interface Definition Language.

**IDL compiler**. RPC: A compiler that processes an RPC interface definition and an optional attribute configuration file (ACF) to generate client and server stubs, and header files See *Interface Definition Language*.

**implicit binding method**. RPC: The implicit method of managing the binding for a remote procedure call in which a global variable in the client application holds a binding handle that the client stub passes to the RPC runtime. See *automatic binding method*, *binding handle*, and *explicit binding method*.

**import**. (1) RPC: To obtain binding information from a name service database about a server that offers a given RPC interface by calling the RPC NSI import operation. (2) RPC: To incorporate constant, type, and import declarations from one RPC interface definition into another RPC interface definition by means of the IDL import statement.

**import context**. The context set up by the client to import compatible binding handles from the name space. Name service interfaces (NSI) are used to set up and free the import context.

**inaccuracy**. DTS: The bounded uncertainty of a clock value as compared to a standard reference.

**instance**. XOM: An object in the category represented by a class.

**instance UUID**. RPC: An object Universal Unique Identifier (UUID) that is associated with a single server instance and is provided to clients to identify that instance unambiguously. See *object UUID* and *server instance*.

**integrity**. RPC: A protection level that may be specified in secure RPC communications to ensure that data transferred between two principals has not been changed in transit.

**interface**. RPC: A shared boundary between two or more functional units, defined by functional characteristics, signal characteristics, or other characteristics, as appropriate. The concept includes the specification of the connection of two devices having different functions. See *RPC interface*.

**interface definition**. RPC: A description of an RPC interface written in the DCE Interface Definition Language (IDL). See *RPC interface*.

**Interface Definition Language (IDL)**. A high-level declarative language that provides syntax for interface definitions.

**interface handle**. RPC: A reference in code to an interface specification. See *binding handle* and *interface specification*.

**interface identifier**. RPC: A string containing the interface Universal Unique Identifier (UUID) and major and minor version numbers of a given RPC interface. See *RPC interface*.

**interface specification**. RPC: An opaque data structure that is generated by the DCE IDL compiler from an interface definition. It contains identifying and descriptive information about an RPC interface. See *interface definition*, *interface handle*, and *RPC interface*.

**interface UUID**. RPC: The Universal Unique Identifier (UUID) generated for an RPC interface definition using the UUID generator. See *interface definition* and *RPC interface*.

**International Organization for Standardization (ISO)**. An international body composed of the national standards organizations of 89 countries. ISO issues standards on a vast number of goods and services including networking software.

**Internet address**. The 32-bit address assigned to hosts in a TCP/IP network.

**Internet Protocol (IP)**. In TCP/IP, a protocol that routes data from its source to its destination in an Internet environment. IP provides the interface from the higher level host-to-host protocols to the local network protocols. Addressing at this level is usually from host to host.

**interoperability**. The capability to communicate, execute programs, or transfer data among various functional units in a way that requires the user to have little or no knowledge of the unique characteristics of those units.

**interval**. DTS: The combination of a time value and the inaccuracy associated with it; the range of values represented by a combined time and inaccuracy notation. As an example, the interval 08:00.00I00:05:00 (eight o'clock, plus or minus five minutes) contains the time 07:57.00.

**IP**. Internet Protocol

**ISO**. International Organization for Standardization

## J

**junction**. A specialized entry in the DCE namespace that contains binding information to enable communications between different DCE services.

## K

**Kerberos**. The authentication protocol used to carry out DCE private key authentication. Kerberos was developed at the Massachusetts Institute of Technology.

**key**. A value used to encrypt and decrypt data.

**key file**. A file that contains encryption keys for noninteractive principals.

**key management facility**. A Security Service facility that enables noninteractive principals to manage their secret keys.

## L

**LAN**. Local area network.

**layer**. In network architecture, a group of services, functions, and protocols that is complete from a conceptual point of view, that is one out of a set of hierarchically arranged groups, and that extends across all systems that conform to the network architecture.

**listener thread**. Created by RPC to listen on all TCP/IP sockets for calls coming into the client for the datagram protocol and for calls coming into the server for datagram and connection-oriented protocols.

**little endian**. An attribute of data representation that reflects how multi-octet data is stored. In little endian representation, the lowest addressed octet of a multi-octet data item is the least significant. See *big endian*.

**liveness**. Context handle and related maintenance functions that maintain context on behalf of clients even during periods of nominal client inactivity.

**local**. (1) Pertaining to a device directly connected to a system without the use of a communication line. (2) Pertaining to devices that have a direct, physical connection. Contrast with *remote*.

**local application thread**. RPC: An application thread that runs within the confines of one address space on a local system and passes control exclusively among local code segments. See *application thread*, *client application thread*, *RPC thread* and *server application thread*.

**local area network (LAN)**. A network in which communication is limited to a moderate-sized geographical area (1 to 10 km) such as a single office building, warehouse, or campus, and which does not generally extend across public rights-of-way. A local network depends on a communication medium capable of moderate to high data rate (greater than 1Mbps), and normally operates with a consistently low error rate.

**local cell**. The cell to which the local machine belongs. Synonymous with *home cell*. Contrast with *foreign cell*.

**local file system (LFS)**. An organized collection of data in the form of a root directory and its subdirectories and files. An LFS supports special features useful in a distributed environment: the ability to replicate data; to log file system data, enabling quick recovery after a crash; to simplify administration by dividing the file system into easily managed units called filesets; and to associate access control lists (ACLs) with files and directories. An LFS is located on a disk that is physically attached to a machine In other file systems, a single disk partition contains only one file system. In DCE LFS an aggregate can contain multiple file systems (filesets). See also *access control list (ACL)*.

**local name**. A name that is meaningful and usable only within the cell where an entry exists. The local name is a shortened form of a global name. Local names begin with the prefix /.: and do not contain a cell name. Synonymous with *cell-relative name*.

**local server**. DTS: A server that synchronizes with its peers and provides its clock value to other servers and clerks in the same network.

**local type**. RPC: A type named in a **represent_as** clause and used by application code to manipulate data that is passed in a remote procedure call as a network type. See *network type*.

**logical unit (LU)**. A host port through which a user gains access to the services of a network.

**login facility**. A Security Service facility that enables a principal to establish its identity.

## M

**manager**. RPC: A set of remote procedures that implement the operations of an RPC interface and that can be dedicated to a given type of object. See also *object* and *RPC interface*.

**manager entry point vector**. RPC: The runtime code on the server side uses this entry point vector to dispatch incoming remote procedure calls. See *entry point vector* and *manager*.

**manager thread**. See *call thread*.

**marshalling**. RPC: The process by which a stub converts local arguments into network data and packages the network data for transmission. Contrast with *unmarshalling*.

**mask**. (1) A pattern of characters used to control the retention or deletion of portions of another pattern of characters (2) Security: Used to establish maximum permissions that can then be applied to individual ACL entries. (3) GDS: The administration screen interface menus.

**master replica**. CDS: The first instance of a specific directory in the namespace. After copies of the directory have been made, a different replica can be designated as the master, but only one master replica of a directory can exist at a time. CDS can create, update, and delete object entries and soft links in a master replica.

**maybe semantics**. RPC: A form of idempotent semantics that indicates that the caller neither requires nor receives any response or fault indication for an operation, even though there is no guarantee that the operation was completed. An operation with maybe semantics is implicitly idempotent and lacks output parameters. See *at-most-once semantics*, *broadcast semantics*, and *idempotent semantics*.

**mutex**. Mutual exclusion. A read/write lock that grants access to only a single thread at any one time. A mutex is often used to ensure that shared variables are always seen by other threads in a consistent way.

**mvsexpt**. One of two (the other is **mvsimpt**) utilities used to automate much of the administrator's work in creating the cross-linking information for DCE-RACF interoperability. The **mvsexpt** utility creates the cross-linking information in the RACF database from information in the DCE registry. See also *cross-linking information*, *interoperability*, and *single sign-on*.

**mvsimpt**. One of two (the other is **mvsexpt**) utilities used to automate much of the administrator's work in creating the cross-linking information for DCE-RACF interoperability. The **mvsimpt** utility creates DCE principals from information obtained from the RACF database. See also *cross-linking information*, *interoperability*, and *single sign-on*.

**name**. GDS, CDS: A construct that singles out a particular (directory) object from all other objects. A name must be unambiguous (denote only one object); however, it need not be unique (be the only name that unambiguously denotes the object).

**name service**. A central repository of named resources in a distributed system. In DCE, this is the same as Directory Service.

**name service handle**. RPC: An opaque reference to the context used by the series of next operations called during a specific name service interface (NSI) search or inquiry.

**name service interface (NSI)**. RPC: A part of the application program interface (API) of the RPC run time. NSI routines access a name service, such as CDS, for RPC applications.

**namespace**. CDS: A complete set of CDS names that one or more CDS servers look up, manage, and share. These names can include directories, object entries, and soft links.

**NCA**. Network Computing Architecture.

**NDR**. Network Data Representation.

**network**. A collection of data processing products connected by communications lines for exchanging information between stations.

**network address**. An address that identifies a specific host on a network. Synonymous with *host ID*.

**Network Computing Architecture (NCA)**. RPC: An architecture for distributing software applications across heterogeneous collections of networks, computers, and programming environments using UDP. NCA specifies part of the DCE Remote Procedure Call architecture.

**network data**. RPC: Data represented in a format defined by a transfer syntax. See also *transfer syntax*.

**Network Data Representation (NDR)**. RPC: The transfer syntax defined by the Network Computing Architecture. See *transfer syntax*.

**network descriptor**. RPC: The identifier of a potential network channel, such as a UNIX socket.

**network protocol**. A communications protocol from the Network Layer of the Open Systems Interconnection (OSI) network architecture, such as the Internet Protocol (IP).

**Network Time Protocol (NTP)**. A clock synchronization protocol commonly used on an Internet.

**network type**. RPC: A type defined in an interface definition and referenced in a **represent_as** clause that is converted into a local type for manipulation by application code. See *local type*.

**node**. (1) An endpoint of a link, or a junction common to two or more links in a network. Nodes can be preprocessors, controllers, or workstations, and they can vary in routing and other functional capabilities. (2) In network topology, the point at an end of a branch. It is usually a physical machine.

**non-idempotent**. An RPC call attribute type describing an RPC call that must run no more than once. Before running a non-idempotent call, servers and clients verify each other's identity using one of the simple conversation callback operations provided by a set of conversation manager routines for the datagram RPC protocol service.

**null time provider**. The daemon that fetches the time from the hardware clock of the DCE host for DTS.

**NSI**. Name service interface.

**NSI attribute**. RPC: An RPC-defined attribute of a name service entry used by the RPC name service interface. A name service interface (NSI) attribute stores one of the following: binding information, object Universal Unique Identifiers (UUIDs), a group, or a profile. See *NSI binding attribute*, *NSI group attribute*, *NSI object attribute*, and *NSI profile attribute*.

**NSI binding attribute**. RPC: An RPC-defined attribute (NSI attribute) of a name service entry; the binding attribute stores binding information for one or more interface identifiers offered by an RPC server and identifies the entry as an RPC server entry. See *binding information* and *NSI object attribute*. See also *server entry*.

**NSI group attribute**. RPC: An RPC-defined attribute (NSI attribute) of a name service entry that stores the entry names of the members of an RPC group and identifies the entry as an RPC group. See *group*.

**NSI object attribute**. RPC: An RPC-defined attribute (NSI attribute) of a name service entry that stores the object UUIDs of a set of RPC objects. See *object*.

**NSI profile attribute**. RPC: An RPC-defined attribute (NSI attribute) of a name service entry that stores a collection of RPC profile elements and identifies the entry as an RPC profile. See *profile*.

**NTP**. Network Time Protocol.

**NULL**. In the C language, a pointer that does not point to a data object.

## O

**object**. (1) A data structure that implements some feature and has an associated set of operations. (2) RPC: For RPC applications, anything that an RPC server defines and identifies to its clients using an object Universal Unique Identifier (UUID). An RPC object is often a physical computing resource such as a database, directory, device, or processor. Alternatively, an RPC object can be an abstraction that is meaningful to an application, such as a service or the location of a server. See *object UUID*. (3) XDS: Anything in the world of telecommunications and information processing that can be named and for which the directory information base (DIB) contains information. (4) XOM: Any of the complex information objects created, examined, changed, or destroyed by means of the interface.

**object class table (OCT)**. A recurring attribute of the directory schema with the description of the object classes permitted.

**object entry**. CDS: The name of a resource (such as a node, disk, or application) and its associated attributes, as stored by CDS. CDS administrators, client application users, or the client applications themselves can give a resource an object name. CDS supplies some attribute information (such as a creation timestamp) to become part of the object, and the client application may supply more information for CDS to store as other attributes. See *entry*.

**object identifier (OID)**. A value (distinguishable from all other such values) that is associated with an information object. It is formally defined in the CCITT X.208 standard.

**object management (OM)**. The creation, examination, change, and deletion of potentially complex information objects.

**object name**. CDS: A name for a network resource.

**object UUID**. RPC: The Universal Unique Identifier (UUID) that identifies a particular RPC object. A server specifies a distinct object UUID for each of its RPC objects. To access a particular RPC object, a client uses the object UUID to find the server that offers the object. See *object*.

**octal**. In reference to a selection, choice or condition that has eight possible different values or states. In reference to a fixed-radix numeration having a radix of eight.

**OCT**. Object class table.

**octet**. A byte that consists of eight bits.

**opaque**. A datum or data type whose contents are not visible to the application routines that use it.

**Open Software Foundation (OSF)**. A nonprofit research and development organization set up to encourage the development of solutions that allow computers from different vendors to work together in a true open-system computing environment.

**open system**. A system whose characteristics comply with standards made available throughout the industry and that can be connected to other systems complying with the same standards.

**open systems interconnection (OSI)**. The interconnection of open systems in accordance with standards of the International Organization for Standardization (ISO) for the exchange of information.

**operation**. (1) GDS: Processing performed within the directory to provide a service, such as a read operation. (2) RPC: The task performed by a routine or procedure that is requested by a remote procedure call.

**organization**. (1) The third field of a subject identifier. (2) Security: Data that associates a named set of users who can be granted common access rights that are usually associated with administrative policy.

**orphaned call**. RPC: A call running in an RPC server after the client that started the call fails or loses communication with the server.

**OSF**. Open Software Foundation.

**OSI**. Open systems interconnection

## P

**PAC**. Privilege attribute certificate.

**package**. XOM: A specified group of related object management (OM) classes, denoted by an object identifier.

**packet**. (1) In data communication, a sequence of binary digits, including data and control signals, that is transmitted and switched as a composite whole. [1] The data, call control signals, and error control information are arranged in a specific format. (2) See *call-accepted packet, call-connected packet, call-request packet*. See *clear-confirmation packet, clear-indication packet, clear-request packet*. See *data packet, incoming-call packet.*

**parent directory**. CDS: Any directory that has one or more levels of directories beneath it in a cell name space. A directory is the parent of any directory immediately beneath it in the hierarchy.

**parent process**. A process created to carry out a program. The parent process in turn creates child processes to process requests. Contrast with *child process*.

**partially bound binding handle**. RPC: A server binding handle that contains an incomplete server address lacking an endpoint. Contrast with *fully bound binding handle*.

**Partitioned data set (PDS)**. A data set in direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data.

**password**. A secret string of characters shared between a computer system and a user. The user must specify the character string to gain access to the system.

**PCS**. Portable Character Set.

**peer trust**. A type of trust relationship established between two cells by means of a secret key shared by authentication surrogates maintained by the two cells. A peer trust relationship enables principals in one cell to communicate securely with principals in the other.

**permission**. (1) The modes of access to a protected object. The number and meaning of permissions with respect to an object are defined by the access control list (ACL) Manager of the object. (2) GDS: One of five groups that assigns modes of access to users: MODIFY PUBLIC, READ STANDARD, MODIFY STANDARD, READ SENSITIVE, or MODIFY SENSITIVE. Synonymous with *access right*. See also *access control list*.

**person**. See *principal*.

**pickle**. A type of data encoding. When a Remote Procedure Call (RPC) sends data between a client and a server, it serializes the user's data structures by using the IDL Encoding Services (ES). This serialization scheme for encoding and decoding data is informally called *pickling*,

**ping**. Utility in TCP/IP which is used to test if a destination host can be reached by sending test packets and waiting for a reply.

In the RPC control program, a command to test if a server is listening.

**pipe**. (1) RPC: A mechanism for passing large amounts of data in a remote procedure call. (2) The data structure that represents this mechanism.

**plaintext**. The input to an encryption function or the output of a decryption function. Encryption transforms plaintext to ciphertext and decryption transforms ciphertext into plaintext.

**platform**. The operating system environment in which a program runs.

**port**. (1) Part of an Internet Protocol (IP) address specifying an endpoint. (2) To make the programming changes necessary to allow a program that runs on one type of computer to run on another type of computer.

**Portable Character Set**. A set of characters to enable internationalization. A character set used by DCE to enable word wide connectivity by ensuring that a minimum group of characters is supported in DCE. All DCE RPC clients and servers are required to support the DCE PCS.

**position (within a string)**. XOM: The ordinal position of one element of a string relative to another.

**position (within an attribute)**. XOM: The ordinal position of one value relative to another.

**potential binding**. RPC: A specific combination of an RPC protocol sequence, RPC protocol major version, network address, endpoint, and transfer syntax that an RPC client can use to establish a binding with an RPC server. See *binding*. See also *endpoint*, *network address*, *RPC protocol*, *RPC protocol sequence*, and *transfer syntax*.

**predicate**. A Boolean logic term denoting a logical expression that determines the state of some variables. For example, a predicate can be an expression stating that variable A must have the value 3. The control expression used in conjunction with condition variables is based on a predicate. A condition variable can be used to wait for some predicate to become true, for example, to wait for something to be in a queue.

**presentation address**. An unambiguous name that is used to identify a set of presentation service access points. Loosely, it is the network address of an open systems interconnection (OSI) service.

**presented type**. RPC: For data types with the Interface Definition Language (IDL) **transmit_as** attribute, the data type that clients and servers manipulate. Stubs invoke conversion routines to convert the presented type to a transmitted type, which is passed over the network. See *transmitted type*.

**primitive binding handle**. RPC: A binding handle whose data type in Interface Definition Language (IDL) is **handle_t** and in application code is **rpc_binding_handle_t**. See *customized binding handle*.

**principal**. Security: An entity that can communicate securely with another entity. In the DCE, principals are represented as entries in the Registry database and include users, servers, computers, and authentication surrogates.

**privacy**. RPC: A protection level that encrypts RPC argument values. in secure RPC communications.

**private key**. See *secret key*.

**privilege attribute**. Security: An attribute of a principal that may be associated with a set of permissions. DCE privilege attributes are identity-based and include the principal’s name, group memberships, and local cell.

**privilege attribute certificate (PAC)**. Security: Data describing a principal’s privilege attributes that has been certified by an authority. In the DCE, the Privilege Service is the certifying authority; it seals the privilege attribute data in a ticket. The authorization protocol, DCE Authorization, determines the permissions granted to principals by comparing the privilege attributes in PACs with entries in an access control list.

**privilege service**. Security: One of three services provided by the Security Service; the Privilege Service certifies a principal’s privileges. The other services are the Registry Service and the Authentication Service.

**privilege ticket**. Security: A ticket that contains the same information as a simple ticket, and also includes a privilege attribute certificate. See *service ticket*, *simple ticket*, and *ticket-granting ticket*.

**procedure declaration**. RPC: The syntax for an operation, including its name, the data type of the value it returns (if any), and the number, order, and data types of its parameters (if any).

**product-sensitive programming interface (PSPI)**. (1) A special interface that is intended only to be used for a specialized task such as diagnosis, modification, monitoring, repairing, tailoring, or tuning. (2) A special interface that is dependent on or requires the customer to understand significant aspects of the detailed design and implementation of the IBM software product.

**profile**. RPC: An entry in a name service database that contains a collection of elements from which name service interface (NSI) search operations construct search paths for the database. Each search path is composed of one or more elements that refer to name service entries corresponding to a given RPC interface and, optionally, to an object. See *NSI profile attribute* and *profile element*.

**profile element**. RPC: A record in an RPC profile that maps an RPC interface identifier to a profile member (a server entry, group, or profile in a name service database). See *profile*. See also *group*, *interface identifier* and *server entry*.

**profile member**. RPC: A name service entry whose name occupies the member field of an element of the profile. See *profile*.

**programming interface**. The supported method through which customer programs request software services. The programming interface consists of a set of callable services provided with the product.

**proprietary**. Pertaining to the holding of the exclusive legal rights in making, using, or marketing a product.

**protection level**. The degree to which secure network communications are protected. Synonymous with *authentication level*.

**protocol**. A set of semantic and syntactic rules that determines the behavior of functional units in achieving communication.

**protocol family**. Synonym for *address family*.

**protocol sequence**. Synonym for *RPC protocol sequence*.

**protocol sequence vector**. RPC: A data structure that contains an array-size count and an array of pointers to RPC protocol-sequence strings. See *RPC protocol sequence*.

## R

**RACF**. Resource Access Control Facility.

**read access**. CDS: An access right that grants the ability to view data.

**read-only replica**. (1) CDS: A copy of a CDS directory in which applications cannot make changes. Although applications can look up information (read) from it, they cannot create, change, or delete entries in a read-only replica. Read-only replicas become consistent with other, changeable replicas of the same directory during skulks and routine propagation of updates. (2) Security: A replicated Registry server.

**realm**. Security: A cell, considered exclusively from the point of view of Security; this term is used in Kerberos specifications. The term cell designates the basic unit of DCE configuration and administration and incorporates the notion of a realm.

**reference monitor**. Code that controls access to an object. In the DCE, servers control access to the objects they maintain; for a given object, the ACL manager associated with that object makes authorization decisions concerning the object.

**reference pointer**. RPC: A non-null pointer whose value is invariant during a remote procedure call and cannot point at aliased storage.

**referral**. GDS: An outcome that can be returned by a DSA that cannot perform an operation itself. The referral identifies one or more other DSAs more able to perform the operation.

**register**. (1) RPC: To list an RPC interface with the RPC runtime. (2) To place server-addressing information into the local endpoint map. (3) To insert authorization and authentication information into binding information. See *endpoint map* and *RPC interface*.

**Registry database**. Security: A database of security information about principals, groups, organizations, accounts, and security policies.

**Registry Service**. Security: One of three services provided by the Security Service; the Registry Service manages information about principals, accounts, and security policies. The other services are the Privilege Service and the Authentication Service.

**relative time**. A discrete time interval that is usually added to or subtracted from an absolute time. See *absolute time*.

**remote**. Pertaining to a device, file or system that is accessed by your system through a communications line. Contrast with *local*.

**remote procedure**. RPC: An application procedure located in a separate address space from calling code. See *remote procedure call*.

**remote procedure call**. RPC: A client request to a service provider located anywhere in the network.

**Remote Procedure Call (RPC)**. A DCE component. It allows requests from a client program to access a procedure located anywhere in the network.

**replica**. CDS: A directory in the CDS namespace. The first instance of a directory in the name space is the master replica. See *master replica* and *read-only replica*.

**replication**. The making of a shadow of a database to be used by another node. Replication can improve availability and load-sharing.

**request**. A command sent to a server over a connection.

**request buffer**. RPC: A FIFO queue where an RPC system temporarily stores call requests that arrive at an endpoint of an RPC server until the server can process them.

**resource**. Items such as printers, plotters, data storage, or computer services. Each has a unique identifier associated with it for naming purposes.

**Resource Access Control Facility (RACF)**. An IBM licensed program, that provides for access control by identifying and verifying the users to the system, authorizing access to protected resources, and logging the detected unauthorized access to protected resources.

**return value**. A function result that is returned in addition to the values of any output or input/output arguments.

**ROM**. Read-only memory.

**RPC**. Remote Procedure Call.

**RPC control program (RPCCP)**. An interactive administrative facility for managing name service entries and endpoint maps for RPC applications.

**RPCCP**. RPC control program

**RPC interface**. A logical group of operations, data types, and constant declarations that serves as a network contract for a client to request a procedure in a server. See also *interface definition* and *operation*.

**RPC protocol**. An RPC-specific communications protocol that supports the semantics of the DCE RPC API and runs over either connectionless or connection-oriented communications protocols.

**RPC protocol sequence**. A valid combination of communications protocols represented by a character string. Each RPC protocol sequence typically includes three protocols: a network protocol, a transport protocol, and an RPC protocol that works with the network and transport protocols. See *network protocol*, *RPC protocol*, and *transfer protocol*. Synonymous with *protocol sequence*.

**RPC runtime**. A set of operations that manages communications, provides access to the name service database, and performs other tasks, such as managing servers and accessing security information, for RPC applications. See *RPC runtime library*.

**RPC runtime library**. A group of routines of the RPC runtime that support the RPC applications on a system. The runtime library provides a public interface to application programmers, the application programming interface (API), and a private interface to stubs, the stub programming interface (SPI). See *RPC runtime*.

**RPC thread**. A logical thread within which a remote procedure call is executed. See *thread*.

**rundown procedure**. RPC: A procedure used with a context handle that is called following a communications failure. It recovers resources reserved by a server for servicing requests by a particular client. See *context handle*.

**runtime semantics**. RPC: The rules of run time for a remote procedure call, including the effect of multiple calls on the outcome of a procedure's operation. See *at-most-once semantics*, *broadcast semantics*, *idempotent semantics*, and *maybe semantics*.

## S

**scalability**. The ability of a distributed system to expand in size without changes to the system structure, applications, or the way users deal with the system.

**schema**. See *directory schema*.

**secret key**. Security: A long-lived encryption key shared between a principal and the Authentication Service.

**Security Service**. A DCE component that provides trustworthy identification of users, secure communications, and controlled access to resources in a distributed system.

**segment**. One or more contiguous elements of a string.

**server**. (1) On a network, the computer that contains programs, data, or provides the facilities that other computers on the network can access. (2) The party that receives remote procedure calls. Contrast with *client*.

**server addressing information**. RPC: An RPC protocol sequence, network address, and endpoint that represent one way to access an RPC server over a network; a part of server binding information. See *network address*. See also *binding information*, *endpoint*, and *RPC protocol sequence*.

**server application thread**. RPC: A thread running the server application code that initializes the server and listens for incoming calls. See *application thread*, *client application thread*, *local application thread*, and *RPC thread*.

**server binding information**. RPC: Binding information for a particular RPC server. See *binding information* and *client binding information*.

**server entry**. RPC: A name service entry that stores the binding information associated with the RPC interfaces of a particular RPC server and object Universal Unique Identifiers (UUIDs) for any objects offered by the server. See also *binding information*, *NSI binding attribute*, *NSI object attribute*, *object* and *RPC interface*.

**server instance**. RPC: A server running in a specific address space. See *server*.

**server state**. Application Support Server: The condition of the Application Support Server after it has been started. The server state may be any of the following, depending on the actions directed to it by the administrator: initializing, quiescent, starting, operating, or stopping.

**server stub**. RPC: The surrogate calling code for an RPC interface that is linked with server application code containing one or more sets of remote procedures (managers) that implement the interface. See *client stub*. See also *manager*.

**service**. In network architecture, the capabilities that the layers closer to the physical media provide to the layers closer to the end user.

**service ticket**. Security: A ticket for a specified service other than the ticket-granting service. See *privilege ticket*, *simple ticket*, and *ticket-granting ticket*.

**session**. GDS: A sequence of directory operations requested by a particular user of a particular directory user agent (DUA) using the same session object management (OM) object.

**session key**. Security: A short-lived encryption key provided by the Authentication Service to two principals for the purpose of ensuring secure communications between them. Synonymous with *conversation key*.

**shell script**. A file containing shell commands. If the file can be processed, you can specify its name as a simple command. Processing of a shell script causes a shell to run the commands in the script. Alternatively, a shell can be requested to run the commands in a shell script by specifying the name of the shell script as the operand **sh** utility.

**SID**. Subject identifier.

**signal**. Threads: To wake only one thread waiting on a condition variable. See *broadcast*.

**signed**. Security: Pertaining to information that is appended to an enciphered summary of the information. This information is used to ensure the integrity of the data, the authenticity of the originator, and the unambiguous relationship between the originator and the data.

**sign-on**. (1) A procedure to be followed at a terminal or workstation to establish a link to a computer. (2) To begin a session at a workstation. (3) Same as log on or log in.

**simple ticket**. Security: A ticket that contains the principal's identity, a session key, a timestamp and other information, sealed using the target's secret key. See *privilege ticket*, *service ticket*, and *ticket-granting ticket*.

**single sign-on**. In z/OS DCE, single sign-on to DCE allows a z/OS user who has already been authenticated to an external security manager, such as RACF, to be logged in to DCE. DCE does this automatically when a DCE application is started, if the user is not already logged in to DCE.

**skew**. The time difference between two clocks or clock values.

**socket**. A unique host identifier created by the concatenation of a port identifier with a TCP/IP address.

**specific**. XOM: The attribute types that can appear in an instance of a given class, but not in an instance of its superclasses.

**SPI**. (1) System programming interface. (2) Stub programming interface.

**S-stub**. GDS: The part of the directory system agent (DSA) that establishes the connection to the communications network.

**standard**. A model that is established and widely used.

**string**. An ordered sequence of bits, octets, or characters, accompanied by the string's length.

**stub**. RPC: A code module specific to an RPC interface that is generated by the Interface Definition Language (IDL) compiler to support remote procedure calls for the interface. RPC stubs are linked with client and server applications and hide the intricacies of remote procedure calls from the application code. See *client stub* and *server stub*.

**Stub programming interface (SPI)**. RPC : A private runtime interface whose routines are unavailable to application code.

**subject identifier (SID)**. A string that identifies a user or set of users. Each SID consists of three fields in the form person.group.organization. In an account, each field must have a specific value; in an access control list (ACL) entry, one or more fields may use a wildcard.

**synchronization**. DTS: The process by which a Distributed Time Service entity requests clock values from other systems, computes a new time from the values, and adjusts its system clock to the new time.

**syntax**. (1) XOM: An object management (OM) syntax is any of the various categories into which the OM specification statically groups values on the basis of their form. These categories are additional to the OM type of the value. (2) A category into which an attribute value is placed on the basis of its form. See *attribute syntax*.

**System programming interface (SPI).**. A private interface reserved for use by other services within a system and not available to application code. Contrast with *API*.

**system time**. The time value maintained and used by the operating system.

## T

**TCP**. Transmission Control Protocol

**TCP/IP**. Transmission Control Protocol/Internet Protocol

**TDF**. Time differential factor.

**thread**. A single sequential flow of control within a process.

**thread handle**. RPC: A data item that enables threads to share a storage management environment.

**Threads Service**. A DCE component that provides portable facilities that support concurrent programming. The threads service includes operations to create and control multiple threads of execution in a single process and to synchronize access to global data within an application.

**ticket**. Security: An application-transparent mechanism that transmits the identity of an initiating principal to its target. See *privilege ticket*, *service ticket*, *simple ticket* and *ticket-granting ticket*.

**ticket-granting ticket**. Security: A ticket to the ticket-granting service. See *privilege ticket*, *service ticket*, and *simple ticket*.

**time differential factor (TDF)**. DTS: The difference between universal time coordinated (UTC) and the time in a particular time zone.

**timeout period**. The amount of time in seconds that the Control Task waits for a daemon to initialize successfully. If this time elapses and the daemon does not indicate to the Control Task that it has successfully initialized, the daemon's state is deemed to be UNKNOWN.

**time provider (TP)**. DTS: A process that queries universal time coordinated (UTC) from a hardware device and provides it to the server.

**time provider interface (TPI)**. An interface between the DTS server and external time provider process. The DTS server uses the interface to communicate with the time provider and to obtain timestamps from an external time source.

**time provider program**. DTS: An application that functions as a time provider.

**top level pointer**. RPC: A pointer parameter that in a chain of pointers is the only member that is not the referent of any other pointer.

**tower**. CDS: A set of physical address and protocol information for a particular server. CDS uses this information to locate the system on which a server resides and to determine which protocols are available at the server. Tower values are contained in the **CDS_Towers** attribute associated with the object entry that represents the server in the cell namespace.

**TP**. Time provider.

**TP server**. DTS: A server connected to a time provider (TP).

**TPI**. Time provider interface.

**transaction**. (1) A unit of processing consisting of one more application programs initiated by a single request, often from a terminal. (2) IMS/ESA: A message destined for an application program.

**transfer syntax**. RPC: A set of encoding rules used for transmitting data over a network and for converting application data to and from different local data representations. See also *Network Data Representation*.

**Transmission Control Protocol (TCP)**. A communications protocol used in Internet and any other network following the U.S. Department of Defense standards for inter-network protocol. TCP provides a reliable host-to-host protocol in packet-switched communication networks and in an interconnected system of such networks. It assumes that the Internet Protocol is the underlying protocol. The protocol that provides a reliable, full-duplex, connection-oriented service for applications.

**Transmission Control Protocol/Internet Protocol (TCP/IP)**. A set of non-proprietary communications protocols that support peer-to-peer connectivity functions for both local and wide area networks.

**transmitted type**. RPC: For data types with the IDL **transmit_as** attribute, the data type that stubs pass over the network. Stubs invoke conversion routines to convert the transmitted type to a presented type, which is manipulated by clients and servers. See *presented type*.

**transport independence**. RPC: The capability, without changing application code, to use any transport protocol that both the client and server systems support, while guaranteeing the same call semantics. See *transport layer* and *transport protocol*.

**transport layer**. A network service that provides end-to-end communications between two parties, while hiding the details of the communications network. The Transmission Control Protocol (TCP) and International Organization for Standardization (ISO) TP4 transport protocols provide full-duplex virtual circuits on which delivery is reliable, error free, sequenced, and duplicate free. User Datagram Protocol (UDP) provides no guarantees. The connectionless RPC protocol provides some guarantees on top of UDP.

**transport protocol**. A communications protocol, such as the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP).

**trust peer**. One side of a cross-registration that enables two cells to have peer trust. See *peer trust*.

**type**. XOM: A category into which attribute values are placed on the basis of their purpose. See *attribute type*.

**type UUID**. RPC: The Universal Unique Identifier (UUID) that identifies a particular type of object and an associated manager. See also *manager* and *object*.

## U

**UDP**. User Datagram Protocol.

**unexport**. RPC: To remove binding information from a server entry in a name service database. Contrast with *export*.

**Universal Time Coordinated (UTC)**. The basis of standard time throughout the world. Synonymous with Greenwich mean time (GMT).

**Universal Unique Identifier (UUID)**. RPC: An identifier that is immutable and unique across time and space. A UUID can uniquely identify an entity such as an object or an RPC interface. See *interface UUID*, *object UUID*, and *type UUID*.

**unmarshalling**. RPC: The process by which a stub disassembles incoming network data and converts it into local data in the appropriate local data representation. Contrast with *marshalling*.

**user**. A person who requires the services of a computing system.

**User Datagram Protocol (UDP)**. In TCP/IP, a packet-level protocol built directly on the Internet protocol layer. UDP is used for application-to-application programs between TCP/IP host systems.

**UTC**. Universal Time Coordinated

**UUID**. Universal unique identifier

## V

**value**. XOM: An arbitrary and complex information item that can be viewed as a characteristic or property of an object. See *attribute value*.

**varying array**. RPC: An array in which part of its contents is transmitted during a remote procedure call.

**vector**. RPC: An array of references to other structures.

**vendor**. Supplier of software products.

**Virtual Telecommunications Access Method (VTAM)**. An IBM licensed program that controls communication and the flow of data in an SNA network. It provides single-domain, multiple-domain, and interconnected network capability.

## W

**well-known endpoint**. RPC: A preassigned, stable endpoint that a server can use every time it runs. Well-known endpoints typically are assigned by a central authority responsible for a transport protocol. An application declares a well-known endpoint either as an attribute in an RPC interface header or as a variable in the server application code. See *dynamic endpoint* and *endpoint*.

**workstation**. A device that enables users to transmit information to or receive information from a computer, for example, a display station or printer.

## X

**X.500**. The CCITT/ISO standard for the open systems interconnection (OSI) application-layer directory. It allows users to register, store, search, and retrieve information about any objects or resources in a network or distributed system.

**XDS**. The X/Open Directory Services API.

**X/Open Directory Services (XDS)**. An application program interface that DCE uses to access its directory service components. XDS provides facilities for adding, deleting, and looking up names and their attributes. The XDS library detects the format of the name to be looked up and directs the calls it receives to either GDS or CDS. XDS uses the XOM API to define and manage its information.

**XOM**. The X/Open OSI-Abstract-Data Manipulation API.

# Bibliography

This bibliography is a list of publications for z/OS DCE and other products. The complete title, order number, and a brief description is given for each publication.

## z/OS DCE Publications

This section lists and provides a brief description of each publication in the z/OS DCE library.

### Overview

- *z/OS DCE Introduction*, GC24-5911

  This book introduces z/OS DCE. Whether you are a system manager, technical planner, z/OS system programmer, or application programmer, it will help you understand DCE and evaluate the uses and benefits of including z/OS DCE as part of your information processing environment.

### Planning

- *z/OS DCE Planning*, GC24-5913

  This book helps you plan for the organization and installation of z/OS DCE. It discusses the benefits of distributed computing in general and describes how to develop plans for a distributed system in a z/OS environment.

### Administration

- *z/OS DCE Configuring and Getting Started*, SC24-5910

  This book helps system and network administrators configure z/OS DCE.
- *z/OS DCE Administration Guide*, SC24-5904

  This book helps system and network administrators understand z/OS DCE and tells how to administer it from the batch, TSO, and shell environments.
- *z/OS DCE Command Reference*, SC24-5909

  This book provides reference information for the commands that system and network administrators use to work with z/OS DCE.
- *z/OS DCE User's Guide*, SC24-5914

  This book describes how to use z/OS DCE to work with your user account, use the directory service, work with namespaces, and change access to objects that you own.

### Application Development

- *z/OS DCE Application Development Guide: Introduction and Style*, SC24-5907

  This book assists you in designing, writing, compiling, linking, and running distributed applications in z/OS DCE.
- *z/OS DCE Application Development Guide: Core Components*, SC24-5905

  This book assists programmers in developing applications using application facilities, threads, remote procedure calls, distributed time service, and security service.
- *z/OS DCE Application Development Guide: Directory Services*, SC24-5906

  This book describes the z/OS DCE directory service and assists programmers in developing applications for the cell directory service and the global directory service.
- *z/OS DCE Application Development Reference*, SC24-5908

  This book explains the DCE Application Program Interfaces (APIs) that you can use to write distributed applications on z/OS DCE.

### Reference

- *z/OS DCE Messages and Codes*, SC24-5912

  This book provides detailed explanations and recovery actions for the messages, status codes, and exception codes issued by z/OS DCE.

## z/OS SecureWay® Security Server Publications

This section lists and provides a brief description of books in the z/OS SecureWay Security Server library that may be needed for z/OS SecureWay Security Server DCE and for RACF® interoperability.

- *z/OS SecureWay Security Server DCE Overview*, GC24-5921

  This book describes the z/OS SecureWay Security Server DCE and provides z/OS SecureWay Security Server DCE information about the z/OS DCE library.

- *z/OS SecureWay Security Server LDAP Client Programming*, SC24-5924

  This book describes the Lightweight Directory Access Protocol (LDAP) client APIs that you can use to write distributed applications on z/OS DCE and gives you information on how to develop LDAP applications.

- *z/OS SecureWay Security Server RACF Security Administrator's Guide*, SA22-7683.

  This book explains RACF concepts and describes how to plan for and implement RACF.

- *z/OS SecureWay Security Server LDAP Server Administration and Use*, SC24-5923

  This book describes how to install, configure, and run the LDAP server. It is intended for administrators who will maintain the server and database.

- *z/OS SecureWay Security Server Firewall Technologies*, SC24-5922

  This book provides the configuration, commands, messages, examples and problem determination for the z/OS Firewall Technologies. It is intended for network or system security administrators who install, administer and use the z/OS Firewall Technologies.

## Tool Control Language Publication

- *Tcl and the Tk Toolkit*, John K. Osterhout, (c)1994, Addison—Wesley Publishing Company.

  This non-IBM book on the Tool Control Language is useful for application developers, DCECP script writers, and end users.

## IBM C/C++ Language Publication

- *z/OS C/C++ Programming Guide*, SC09-4765

  This book describes how to develop applications in the C/C++ language in z/OS.

## z/OS DCE Application Support Publications

This section lists and provides a brief description of each publication in the z/OS DCE Application Support library.

- *z/OS DCE Application Support Configuration and Administration Guide*, SC24-5903

  This book helps system and network administrators understand and administer Application Support.

- *z/OS DCE Application Support Programming Guide*, SC24-5902

  This book provides information on using Application Support to develop applications that can access CICS® and IMS™ transactions.

## Encina Publications

- *z/OS Encina Toolkit Executive Guide and Reference*, SC24-5919

  This book discusses writing Encina applications for z/OS.

- *z/OS Encina Transactional RPC Support for IMS*, SC24-5920

  This book is to help software designers and programmers extend their IMS transaction applications to participate in a distributed, transactional client/server application.

# Index

# Readers' Comments

**z/OS**
**DCE**
**Application Development Guide:**
**Core Components**

**Publication No. SC24-5905-00**

**You may use this form to report errors, to suggest improvements, or to express your opinion on the appearance, organization, or completeness of this book.**

**Date:** ________________________________

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

**Note**

Report system problems to your IBM representative or the IBM branch office serving you.
U.S. customers can order publications by calling the IBM Software Manufacturing Solutions at **1-800-879-2755**.

In addition to using this postage-paid form, you may send your comments by:

| | | | |
|---|---|---|---|
| FAX | 1-607-752-2327 | Internet | pubrcf@vnet.ibm.com |
| IBMLink | GDLVME(PUBRCF) | | |

**Would you like a reply?** ___**YES** ___ **NO** If yes, please tell us the type of response you prefer.

___ Electronic address: ________________________________

___ FAX number: ________________________________

___ Mail: (Please fill in your name and address below.)

| | |
|---|---|
| Name | Address |
| Company or Organization | |
| Phone No. | |

**Readers' Comments**
SC24-5905-00

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

**BUSINESS REPLY MAIL**

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

Department G60
International Business Machines Corporation
Information Development
1701 North Street
ENDICOTT NY 13760-5553

Fold and Tape | **Please do not staple** | Fold and Tape

SC24-5905-00

Cut or Fold
Along Line

Program Number: 5694-A01

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC24-5905-00

*Spine information:*

IBM z/OS DCE Application Development Guide: Core Components